Planning for Behaviour-Based Robotic Assembly: A Logical Framework

Stephen John Scott Cranefield

Ph.D.

University of Edinburgh

1990

Abstract

This thesis describes an approach to robotic assembly planning based on a logic of *plan specification* formulae. These formulae consist of a term representing the plan, a description of the states before and after the plan is performed (using a simple but structured world model), and a set of literals which can be evaluated as Prolog goals to test preconditions and generate terms to appear in the postconditions. The specifications for the atomic actions are given as axioms and planning takes place by attempting to prove the specification for the desired plan, decomposing it by the reverse application of inference rules that state how the specification for a plan can be deduced from those of its subplans. These are derived from the definitions of the temporal operators that are used to construct plans. The initial plan may be partially instantiated due to prior constraints on the form of the plan, and the remaining plan variables and atomic action parameters will be instantiated during the course of planning. The planning strategy is expressed using goal-decomposing 'tactics'. The representation of world states is defined using equational logic and (equational) unification is used to 'match' these *state specifications*.

The form of this logic is designed to meet the requirements of *behaviour-based* assembly systems such as Edinburgh's SOMASS system, which is described in this thesis, and to allow various temporal languages to be used to represent plans, in order to investigate their uses and to develop appropriate planning strategies.

Acknowledgements

I would like to thank my supervisor, Dr. Tim Smithers, for his support and encouragement, Chris Malcolm for providing a focus to my work by developing the SOMASS system, and the following people for their help during the preparation of this thesis: Mike Cameron-Jones and Jim Donnett for proof-reading, and Brian Logan for being a tolerant system manager. I am also grateful to the Association of Commonwealth Universities for financial support.

In addition to those named above, the following present and former members of the department have all contributed to making my time here enjoyable: Mike Jones, Andy Robertson, Jan Paredis, Myra Wilson, Howard Hughes, Prabhas Chongstitvatana, Diana Bental and Jonathan Salmon.

Thanks also to the members of the Edinburgh University Music Society Symphony Orchestra and especially the E.U. Chamber Orchestra for much good music and many good times.

Finally, I would like to thank the University of Otago for being patient over the last few months.

Declaration

I declare that this thesis has been composed by myself and that the work described in it is my own, except where stated otherwise.

Contents

List of Figures

Chapter 1

Introduction

A wide range of research in AI and robotics has been directed towards increasing the autonomy of robotic assembly systems. At one extreme, the work on general-purpose planning systems has focused on the high-level strategic problem of choosing and ordering actions to achieve a desired goal. Other researchers have tackled the difficulties of controlling the detailed manipulations of a robot that must cope with the uncertainties and imperfections of the real world, such as finding stable grasps and achieving the fine fitting of parts. However, although mobile robots have successfully executed high level plans to achieve simple tasks (e.g. the original SRI robot, Shakey [Nilsson 84]), attempts to link planners with on-line assembly controllers have been largely unsuccessful. This is because traditional approaches have relied on explicitly representing and reasoning about the possible errors in the dimensions and positions of the parts and the accuracy of the robot, predicting where problems could arise, and implementing the high-level actions using complex sequences of motions designed to cope with the worst case. With this approach it is an extremely complicated and generally intractable task to generate an executable program from a plan expressed in terms of high-level operators acting in an idealised world.

1.1 Behaviour-Based Programming

Recently a new approach to robot programming has arisen as a reaction against the standard techniques of 'classical robotics'. In order to avoid the complexity

of coordinating and controlling sensing and action via a central world model, some researchers are aiming to achieve efficient and reliable robot control by coordinating sensing and action in a more task-oriented fashion, following the principles learned from the study of animal sensory and motor control systems. In this methodology, which we will refer to as *behaviour-based programming*, the robot's task is described in terms of the desired behaviours that it must exhibit, and these are implemented by developing a collection of robust task-achieving *behavioural modules* that perform their task competently and reliably across a reasonable range of world states in the robot's problem domain.

In the field of mobile robotics, researchers following this paradigm advocate that a prerequisite for intelligent behaviour is having the ability to move around and survive in a dynamic environment. Their work concentrates on developing a hierarchy of behaviours that implement general low-level survival and exploratory skills, with each level increasing the agent's competence to act in its environment. There is therefore no notion of an overall task-oriented goal to be performed — success is measured in terms of the robustness and flexibility of the robot's behaviour in its intended domain.

In assembly robotics the application of the behaviour-based methodology has a rather different flavour. There is a clearly defined goal that must be achieved repeatedly in an efficient and reliable manner. This suggests that the application of AI planning techniques is not only applicable, but also highly desirable. However, the precise manipulations required for fitting parts together increase the problems of coordinating sensing and action, and coping with the inevitable errors in position and form that arise during assembly. This has caused the plans produced for classical assembly systems to be extremely fragile and almost certain to fail when executed. Attempts to remedy this by planning to increasingly greater levels of detail have been largely unsuccessful due to the intractability of uncertainty reasoning.

In contrast, the SOMASS assembly system [Malcolm 87] developed at Edinburgh has shown that the robust performance and task-achieving nature of behavioural modules enable plans created for an idealised, uncertainty-free world to be successfully executed in the real world, provided that the implemented behaviours are well matched to the current type of assembly problem and make good use of the known structure and

constraints of the problem domain. However, the complex nature of the specialised planner used in the SOMASS system suggests that the techniques of traditional AI planning do not extend in a straightforward way to behaviour-based assembly systems. The abstract operators used to represent actions are not expressive enough to describe the more complicated units of behaviour encapsulated within behavioural modules. Planning in terms of behaviours will require representing the domains of competence and effects of the behaviours as pre- and postconditions of the corresponding planning operators. These may involve spatial and geometric reasoning and could not be adequately represented by add and delete lists of assertions as commonly used in planning. More expressive operator descriptions will in turn necessitate the development of more powerful planning strategies. In this thesis we address this issue by proposing a logical framework for planning in behaviour-based assembly systems, which is designed to be flexible enough to allow complex *behavioural operators* to be expressed and to support the development of planning strategies for different types of assembly problem.

1.2 Modelling the SOMASS Planner

The SOMASS planner is quite unlike traditional AI and assembly planners, as there is no explicit notion of state, or representation of the current partially formed plan. The plan is simply represented by a sequence of records, one for each part, which record the results of the (predetermined) decisions concerning the part that the planner makes at the various stages of planning (e.g. the final position of the part in the assembly, the 'put' and 'get' grips, etc.). As the planner generates plans for a specific assembly problem, for which the required plan structure is already known, the order of search and the particular conditions that must be tested at each stage are hard-wired in.

Although the SOMASS system can intuitively be viewed as an optimised version of a generic planner that reasons in terms of abstracted operators representing behavioural modules, it is difficult to make this notion precise due to the planner's specialisation and implicit use of tacit knowledge about the domain and the system's uncertainty-handling strategies. One of the motivations behind this work was to determine how the different hard-wired computations performed by the planner (e.g. to check part stability, grip

clearance, etc.) could be associated with pre- and postcondition descriptions of the SOMASS system behaviours, and to understand how the planner's hierarchical structure could be expressed using this approach.

Once such a high-level operator-based model is developed, the question arises whether this can now be 'compiled-down' to produce an optimised special-purpose planner similar to the existing one. One possibility would be to partially evaluate a general-purpose planner provided with a skeleton plan describing the structure of the Soma assembly, and to apply program transformation techniques to simplify the data structures used.

Such a representation would also allow formal reasoning about the validity and effects of the uncertainty-reducing strategies that are used. For instance, the original strategy used by the system was to 'pad out' the parts by offsetting them from their nominal final destination by a certain precomputed distance, and then to squeeze out any accumulated uncertainty in the parts' positions by 'patting' the assembly together. It would be useful to be able to investigate the relationship between the ideal world plan generated by the higher levels of the planner and the final version which has the offset and patting information added, in order to verify that transformation between the two levels of plan is correct with respect to the padding strategy.

1.3 The Role of Time in Robotic Assembly

Another issue considered in this thesis is the representation of time for the robotic assembly domain. The ordering of tasks and actions is obviously an important part of assembly planning. Planning researchers have employed various formalisms for representing temporal ordering, ranging from sequences or partial orderings of actions to networks of constraints between points or intervals of time. In addition, there have been many models of time and causality proposed in the fields of knowledge representation (AI), semantics and verification of systems and concurrent processes (computer science), and in philosophical logic.

Robotic assembly has its own particular features that influence the choice of temporal representation. It would be desirable for an assembly plan to allow for as much flexibility

in the ordering of actions at run-time as possible. This would enable the on-line system to use 'opportunistic scheduling' [Fox & Kempf 85a] to minimise delays caused by late-arriving parts, etc. Useful ordering and control constructs include operators for nondeterministic choice, potential concurrency, and partial orders satisfying some given action-ordering constraints. However, there is a trade-off between the expressive power of a representational language and the computational complexity of reasoning about it, and it is likely that there is no best representation that is suitable for all robotic assembly tasks. Existing planners and other reasoning systems generally either have a particular, reasonably efficient temporal representation hard-wired in, or they use temporal logics which are very expressive but computationally intractable. It seems that there is a need for a planning system that allows different temporal languages to be defined and used, and which supports the development of appropriate planning strategies for them.

Another consideration in assembly planning is that in some cases, the desired form of an assembly plan may be already partially determined by design constraints on the parts, desired assembly strategies, the current state of the work-cell, or other contingencies that a planner could not be expected to reason about. Therefore, it should be possible to give the planner a 'skeleton plan' with some temporal constraints already specified and other constraints and operator parameter values to be determined during the course of planning.

1.4 A Logical Framework for Robotic Assembly Planning

The above considerations suggest that a flexible planning system is needed to develop and test new planning techniques based on expressive *behavioural operator* declarations and user-defined temporal operators. In order to meet these criteria, the planning framework described in this thesis is based upon the following features:

- A logic of *plan specification* formulae is used to represent the pre- and postconditions of plans and atomic actions. These formulae generalise the traditional operator declarations of STRIPS [Fikes & Nilsson 71] style planners, although a simpler but more structured world model is used. Planning takes place by

attempting to prove a goal plan specification (describing the initial conditions, desired plan, and final goal condition) by decomposing it into subgoals through the reverse application of inference rules which describe how the specifications for compound plans can be deduced from the specifications for their subplans. Each of these inference rules corresponds to a particular temporal operator and applying the rule results in that operator being introduced into the plan. The specifications of the atomic actions (representing robot behaviours) are supplied by the user as axioms.

The term representing the desired plan may be a variable or a partially instantiated *skeleton plan* due to prior constraints on the form of the plan. The remaining plan variables and atomic action parameters will be instantiated during the course of planning.

- A plan specification formula may include a set of Prolog goals which are used to test complicated preconditions and to construct new terms that appear in the postconditions. The planner can execute these goals at any stage of planning, under the control of the planning strategy.

- The planning process is controlled by using goal-decomposing *tactics* to guide the proof search. This allows specific planning strategies for particular problem domains or temporal languages to be developed.

The semantics of plan specifications include an implicit form of 'frame axiom', stating that any properties that hold before a plan is executed will continue to hold afterwards unless contradicted by the postconditions of the plan specification.

The world is modelled as consisting of a set of *entities* of various types (e.g. objects, places, etc.), each with a number of *attributes* (e.g. location and orientation) that will have values assigned to them. This is a very simple but general representation and is based on the view that planning, in general, is an intractable problem unless the planner's view of the world is carefully structured to reflect the conceptual structure of the problem domain and the agent's competences. Until automatic problem-solvers can perform this structuring themselves, we must provide the appropriate abstractions.

The use of this representation was motivated by analysing the SOMASS planner to determine how, despite its minimal and problem-specific representation, its operations could be viewed as planning in terms of behavioural operators.

Conditions on world states are represented as sets of entity–attribute–value triples, standing for conjunctions of expressions of the form

$$attribute(entity) = value$$

These sets are called *state specifications* as they partially describe states.

The inference process works by resolving the temporal operator 'introduction' rules with the current proof state to produce a new proof state. This requires unifying plan specification formulae. However, the unification process must take into account the equational properties of sets (e.g. $\{x, y\} = \{y, x\}$), and as the state specifications in the conclusions of inference rules may be denoted by (meta-level) functional expressions, it is also necessary to solve equations as part of the unification process. It is therefore important to identify techniques for implementing these processes efficiently.

We have chosen to define state specifications as abstract data types in *order-sorted equational logic*, with conditional rewrite rules used to implement their associated functions and predicates. The technique of *moded equational unification* is used to restrict the application of the unification procedure so that it will only be applied to pairs of state specification terms that can be unified (relatively) efficiently. This also helps to prevent the eager instantiation of variables by the low level computational machinery. This could pre-empt decisions that should be made by the strategic planning component, and would therefore lead to less focused search.

1.5 The Scope of the Thesis

The preceding discussion gave various motivations for the form of our logical framework, in terms of the facilities it should provide and techniques it is designed to support. However, in this thesis we concentrate on developing the theoretical foundations and the implementation techniques required for this method, and do not investigate these

issues any further. In particular, apart from a brief discussion in Chapter 8, we only consider linear planning using a simple sequential composition temporal operator.

1.6 Outline

The rest of this thesis is organised as follows:

Chapter 2 presents a survey of the methods used for representing and reasoning about time, causation and processes in AI, computer science and philosophy, and identifies the important issues and concepts in this field.

Chapter 3 outlines the philosophy and motivations behind the behaviour-based approach to robot programming, and gives an overview of the SOMASS assembly system.

In Chapter 4, the planner in the SOMASS system is discussed in more detail and it is shown how its operation can be described in terms of behavioural operators. This motivates the form of the state specifications and plan specification formulae which are introduced in this chapter.

The semantics of the plan specification logic are presented in Chapter 5, and it is shown how plan generation in this framework corresponds to proving a theorem expressing the desired results of executing the plan.

Chapter 6 presents the theory and computational techniques needed to produce a viable implementation of this planning framework.

Chapter 7 gives further details about these techniques and discusses how they are combined and used in the implemented planner.

Chapter 8 describes the planning process in more detail and suggests various extensions of the framework.

Finally, in Chapter 9 we present a summary of the work, discuss related approaches and offer suggestions for further research.

Chapter 2

A Survey of Representations for Time and Action

A large number of formalisms have been developed for representing and reasoning about actions occurring over a period of time. This work comes mainly from three disciplines:

Philosophical logic: Many philosophical problems involve time. To enable a formal treatment of these, philosophers have developed various modal temporal logics. This type of logic is discussed in Section 2.3.

Artificial Intelligence: Representing and reasoning about time is important in planning (where performing actions can change the state of the world), understanding natural language with complicated tense expressions, and capturing human commonsense reasoning.

Computer Science: It is important to be able to reason formally about large and complicated computer programs or systems. It may be necessary to verify that the performance of a program over time satisfies certain desirable properties. There are many logics and models that have been designed for this purpose, especially for concurrent and communicating processes.

2.1 The Underlying Model of Time

The most fundamental question for any formalism for temporal representation and reasoning is which temporal entities should be taken as primitive, and how they are structured. Our intuitive notion of time suggests that units of time should be (at least) partially ordered. Beyond this there are two main distinctions that can be made.

Point vs. interval representations

The basic unit of time can be considered to be a point (an instantaneous moment in time) or an interval of time. This distinction may be important in some applications. However, this is usually more a matter of convenience as points can be represented as degenerate intervals, and intervals can be modelled by specifying the endpoints if the distinction between open and closed intervals is not vital. It is also possible to represent both points and intervals as basic entities at the cost of introducing more primitive binary relations between units of time.

The most important consideration in choosing between a point and or an interval model is probably the nature of the temporal constraints that need to be represented. If these are predominantly relationships between intervals then an interval model is obviously more efficient, while a point-based model is better if the constraints mostly relate points.

Linear vs. branching time

Time can be seen as having a linear structure where there is only one past and future, or as having a branching structure. Usually time is only allowed to branch into the future (corresponding to alternate future paths) although there may be problems where the past is uncertain and alternate histories need to be represented. The appropriate choice depends on the intended application. For example, in Computer Science a temporal logic formula can be used to specify a desired property of a computer program. Usually only properties that must hold for all possible computations are of interest. Therefore a linear time logic is applicable, as every possible computation can be seen as a linear sequence of actions. In some cases, however, it may be necessary to specify that

there exists a particular computation path to deal with some potential problem. In this situation a branching time logic is needed.

2.2 Types of Representation

There are several methods that can be used for representing and reasoning about orderings of actions.

- *Networks of constraints*

  In this method the actions, events or time units that are of interest are represented as nodes in a network. The temporal relations or constraints are represented as arcs linking the relevant nodes. This is the method generally used in AI planners.

- *Temporal logics*

  These are discussed in Section 2.3.

- *Real or abstract programming languages*

  A programming language has an implicit temporal ordering of the program's statements, defined by the semantics of the language. However, reasoning about the behaviour of the program usually requires a logic with its own semantic model of time.

  Computer scientists have developed a number of abstract computer languages to study formally the specification, verification and analysis of concurrent and communicating systems. These are designed to abstract away from the less important details, leaving only the issues of synchronisation, communication and nondeterminism. These languages typically include operators representing sequencing, concurrent composition, nondeterministic choice and communication. An example of this is the Calculus of Communicating Systems (CCS) [Milner 80].

- *Formal language theory and automata*

  A sequence of atomic actions can be represented as a string of symbols (or a *word* in formal language theory). More generally, the possible computations of a process can be seen as generating a set of words, i.e. a language. A

language may be finitary (consisting of finite words only) or infinitary. In formal language theory there are a number of ways to specify a language. A restricted class of languages can be represented syntactically by regular expressions. In general, there are two non-syntactic methods for specifying a language: the generative approach (using a grammar) and the recognition approach (using an automaton — an abstract machine).

For applications where branching time is necessary, languages are inadequate to represent the possible computations of a process. In this case a computation can be represented by a tree and there are equivalent notions of regular branching expressions and automata that accept trees.

A common technique for giving semantics to a formal model of concurrent computation is to specify rules that assign a nondeterministic automaton to each process modelled. This automaton can then be used to simulate the behaviour of the process.

Of course, these types of model are all related. For example, temporal logic may be used to define the semantics of a network structure and the deductions that can be made, while formal language theory may be used to give the semantics of temporal logic.

2.3 Temporal Logics

Temporal logics can be classified into three categories:

1. 'Naive' First Order Temporal Logics

   With the "naive first order treatment of time" [Reichgelt 87], an ordinary first order logic is modified by adding one or more extra arguments to every predicate whose truth value could vary with time. The extra argument(s) refer to the time at which the formula is supposed to be true. There are several problems with referring to time in this way. Shoham (1986) argues that this method "accords no special status to time — neither conceptual nor notational", which is not appropriate for a logic of time. Another problem is that it is not possible to say (for example) "John will marry Sue" without explicitly mentioning the time

at which this will be true. Instead, this sentence must be represented by a formula like:

$$\exists t > NOW \wedge marries(John, Sue, t)$$

This can lead to clumsy and unnatural expressions compared to the other types of temporal logic. Also, this method is less expressive than the others because it does not allow the set of entities to change from one time to another.

2. Modal Temporal Logics

A modal logic is one where there are different *possible worlds* in which any formula can be interpreted. The truth value of a proposition or the value of a function may depend on which possible world is being considered. The semantics of a modal logic is based on an *accessibility relation* which determines which worlds can be reached from which other worlds. This relation must be reflexive, i.e. every world must be considered to be accessible from itself. As well as standard truth functional connectives, a modal logic has (at least) two modal operators, intuitively corresponding to necessity (true in all accessible worlds) and possibility (true in at least one accessible world). See [Hughes & Cresswell 68] for an introduction to modal logic.

A modal logic can be used to reason about changes occurring over time, where the possible worlds correspond to units of time, and the accessibility relation encodes the temporal ordering. Modal temporal logics are usually based on either linear or branching time. In the more common linear case, the necessity and possibility operators correspond to 'always' and 'eventually'. Other common operators include 'next instant' (if time is modelled as being discrete) and the analogues of always and eventually for referring to the past.

See [Rescher & Urquhart 71] and [van Benthem 82] for an overview of modal temporal logics.

Modal temporal logics are now widely used in Computer Science for specification and verification of programs (see Section 2.5.3), and are beginning to be used in AI. However, modal logic theorem provers are in general inefficient, and this limits their use in AI at present.

3. Reified Temporal Logics

   In a reified logic, "what would otherwise be propositions or formulas, actually appear as arguments to some predicate, say *TRUE*, as in $TRUE(t_1, t_2, color(house, red))$" [Shoham 86]. This type of temporal logic is common in AI, and the logics of Allen (1984) and McDermott (1982) fall into this category. Reichgelt (1987) points out that a reified temporal logic can be viewed as a meta-language, which is used to reason about the object-level formulae of interest. This meta-language can encode the semantics of a modal temporal logic, and in this way the natural expressiveness of modal logics can be combined with the computational advantages of a first order logic. This also means that this type of logic is more expressive than the other varieties because it allows quantification over object-level propositions, and therefore, general laws such as 'causes must precede their effects' can be expressed.

2.4 AI Representations

2.4.1 Representations of Plans

Planning is an area of AI concerned with the problem of choosing and ordering a set of actions to achieve a given goal from a specified initial state. This involves creating, modifying and refining partial plans, which consist of a collection of actions together with some ordering constraints. Actions are usually modelled as operators that act on the world by adding and deleting facts. Each action also has a list of preconditions that specify when it can be applied.

The way plans are represented has important consequences for the type of problems that can be solved and the efficiency of the planner. The methods that have been used include the following:

Linear sequences of actions (e.g. STRIPS [Fikes & Nilsson 71])

With this method any two actions must be ordered even if they are completely independent of each other. As the correct ordering of two actions may not be

discovered until a later stage of planning, the planner must use backtracking or some other search technique to explore the possible sequences of actions.

Partial orders of actions (e.g. NOAH [Sacerdoti 77], NONLIN [Tate 76])

A non-linear planner represents the plan as a network with arcs describing a partial ordering on the actions. This allows actions to be unconstrained relative to each other, and should therefore eliminate some of the backtracking that can arise in a linear planner from actions being incorrectly ordered. This seems to be a more efficient way of planning, although this has never been proven.

Another advantage of using partial orders is that they can represent plans where some actions may be executed concurrently. This can have advantages when the plan is executed, not only because of the extra efficiency of concurrent execution, but also because of the extra flexibility allowed in the ordering of actions. However, non-linear planners do not make any distinction between actions which *can* and actions which *must* be executed concurrently.

Hierarchical plans (e.g. ABSTRIPS [Sacerdoti 74], NOAH, NONLIN)

A hierarchical planner represents the plan at more than one level of detail. A plan is constructed at the most abstract level and then planning progresses by filling in the missing details at the lower levels of the hierarchy. This may introduce some conflicts between the effects and preconditions of actions, and these must be resolved. The process continues until a fully detailed plan is created. Various techniques have been used to construct the abstracted problem spaces, but the most common model is that planning takes place in the higher levels using abstracted versions of the given problem-solving operators.

Hierarchical planning is a useful technique when it is possible to determine a general strategy to solve the problem, without considering every detail. It requires a method of representing plans that can express a hierarchy of actions.

Partial orders with duration and start-time windows (e.g. DEVISER [Vere 81])

This is a basically a partial order of actions, but the representation also includes the duration of actions and upper and lower bounds for the start time of each action. Goal conflicts can be resolved not only by ordering actions, but also by compressing the start-time windows.

Other planners based on DEVISER also allow upper and lower bounds on the durations and finish times of actions.

Network of constraints with upper and lower bounds (e.g. Dean's Time Map Manager [Dean 85])

This is a generalisation of the previous method, consisting of a network with nodes representing points in time (these may be start and end points of intervals) and arcs labelled with upper and lower bounds on the time between the two end-points.

This can be seen as describing a network longest path problem [Bell 85] or a simple linear programming problem [Valdes-Perez 86], and can therefore be solved efficiently. However, Dean's TMM is designed to perform temporal projection (i.e. to make future predictions based on the current state of knowledge) using a causal theory, and so incorporates a number of extra features. These include persistences (when a fact is assumed to remain true for a certain length of time once it becomes true) and truth maintenance.

Petri Nets

A Petri net is an abstract nondeterministic machine, based on a directed graph with two types of nodes: places (representing conditions) and transitions (representing actions). A Petri net is executed by playing the 'token game'. This involves moving tokens from places on one side of a transition (representing the preconditions) to the places on the other side (representing the postconditions). See [Reisig 85] for an introduction to Petri Nets.

Drummond (1985, 1986) uses a plan representation based on Petri nets that can express conditional actions and loops. He also uses truth maintenance to incorporate sensory information into the world model and to determine when an agent should

believe that a condition holds.

A lot of work has been done investigating the uses of Petri nets for the design and analysis of systems. This includes techniques for analysing nets, and their use for modelling systems at different levels of abstraction. However, it seems that Petri nets are difficult to use for generating plans, especially in an incremental fashion where constraints are added one at a time.

Opportunistic Scheduling

When an agent is executing a plan, it may be desirable for there to be as much flexibility as possible in the ordering of actions. For instance, a robot performing an assembly task could take advantage of alternative action orderings to continue working despite delays caused by a temporary shortage of parts. Fox and Kempf (1985a) call this opportunistic scheduling because job scheduling is done at run-time to make the most of any opportunities that arise and to avoid possible delays.

For this to be effective, the plan representation should not force the ordering of actions to be any more constrained than necessary. This is especially important for reasoning about tasks like robotic assembly, where many of the temporal constraints arise from spatial considerations. For many assembly tasks, there are temporal constraints that cannot be represented by a partial order. An example is "do A, B, C in any order as long as C is not last".

Fox and Kempf (1985b) suggest a representation that can express all possible assembly sequences for any given task. Constraints are represented by primitive ordering relationships of the form "$X < Y$" and conjunctions, disjunctions and negations of these. If normalised to disjunctive normal form, a constraint expression can be interpreted in two additional ways: as a function mapping each action to a set of permissible successor actions, and as a union of partial orders.

Homem de Mello and Sanderson (1986) present a representation based on and/or graphs that can also express all possible assembly sequences.

Temporal logic over a structured domain

Lansky's (1986, 1990) multi-agent planner GEMPLAN is based on a linear temporal logic that models events, causal relationships, and temporal ordering and simultaneity relationships. The problem domain is structured by dividing it into regions of activity which are assumed to be causally independent except via specially declared causal interfaces, or 'ports'. The simplest type of region is an 'element', within which all events are constrained to be sequentially ordered. Elements may be clustered into 'groups', whose boundaries delimit the scope of causal interactions between events except for those occurring via ports. Every event must belong to exactly one element, but elements may be included in several groups. This model of concurrent activity (called GEM — the **G**roup **E**lement **M**odel) is designed to reduce the complexity of reasoning about causality by enabling the inherent structural properties of the domain to be captured in the representation. The planning goal is expressed by defining (modal) temporal logic constraints on the occurrence and ordering of events, and the plan is then generated by a process of incremental constraint satisfaction, considering each constraint in turn and patching the current plan so that the constraint is satisfied. As constraint satisfaction for first order linear temporal logic is intractable, the planner can use predeclared plan patching techniques corresponding to particular commonly occurring types of constraint.

Situated Action and Reactive Planning

There has recently been much interest in the problem of providing mobile robots and real-time control systems with the means of reacting quickly to unexpected and unpredictable events while still maintaining goal-directed behaviour overall. To achieve timely response to critical events, it is necessary that the robot's actions are not completely predetermined before execution, or subject to time-consuming replanning operations at run-time. Instead, the various possible event occurrences must be explicitly associated with the appropriate robot responses. This makes the robot plan look more like a set of situation–response rules than a sequence (or other ordered structure) of actions, and it is therefore necessary to find some way of coordinating the actions so that progress towards a goal (or goals) can still be made.

Rosenschein and Kaelbling (1986) compile a description of the robot's goals and its knowledge about the world state into a low-level combinatory logic circuit that controls the robot by linking sensory inputs to actuator output signals.

Georgeff and Lansky (1987) represent the robot's beliefs, desires and intentions explicitly at run-time, and these can be used to index robot actions together with possible situations that may arise. Thus the robot's response to a situation may vary depending on its intentional state.

Schoppers' (1987) 'universal plans' index robot actions by situations and goals, and are intended to provide an appropriate response for every possible contingency that the robot may face.

Drummond (1989) describes a technique for analysing a description of operator pre- and postconditions, together with a set of goals, in order to generate *situated control rules*. By projecting possible execution paths forward from the initial state, the *critical choice points* are determined (these are the states in which the execution of an action may cause all goal states to be unreachable) and for every such choice point, the set of all 'safe' actions (or sets of causally independent actions) are found. Indexing this set by the (relevant) conditions that define the choice point produces a rule that can be used to guide the run-time execution of actions based on local state information.

Agre and Chapman (1987, 1990) present a rather different approach to implementing flexible and timely behaviour in dynamic domains. Their system, Pengi, plays a video game (in real time) by "improvising" its responses to situations as a result of the interaction between hard-wired control rules and a set of procedurally-encoded visual control routines that recognise and search for simple visual patterns under the guidance of the control component.

An interesting debate on the merits of these approaches to planning appears in the AI Magazine, 10(4), 1989.

2.4.2 Logics of Time and Action

The two best known logics for reasoning about time and action are those of McDermott (1982) and Allen (1984). These are both reified temporal logics.

McDermott's temporal logic

The underlying structure of time in McDermott's logic is a dense, partial ordering of states that branches into the future. States are "instantaneous snapshots of the universe" and are arranged into chronicles, each of which is a complete possible history of events that extends infinitely in time. Each state has an associated date. This is a real number representing the time of that state in the chronicles that contain it. McDermott uses this logic to discuss causality, persistence of effects, continuous change and planning.

Allen's interval logic

This is a temporal logic where the basic unit of time is an interval. There are thirteen primitive binary relations between the times of two intervals. These are: before, meets, overlaps, equals, during, starts (i.e. one interval is an initial subinterval of the other), finishes, and their inverses. Any binary relationship between two intervals can be represented as a disjunction of primitive relations. This can be used for planning by keeping a network of constraints with nodes representing intervals and arcs labelled with the primitive binary relations that hold between the two endpoints. As new constraints are added, their effects can be propagated through the network. However, Vilain and Kautz (1986) claim that maintaining consistency in such a network is an NP-hard problem.

2.5 Temporal Representation in Computer Science

The semantics and analysis of concurrent processes is an important issue in Computer Science at present. There are a number of problems associated with concurrent computing that do not arise with sequential execution (e.g. deadlock), and because of this, computer scientists have developed many formal models to help them reason about the behaviour of concurrent systems. This involves verifying properties concerning the occurrence of states and actions over time, and is therefore a form of temporal reasoning.

The introduction to [Brookes, Roscoe & Winskel 84] gives a good overview of work in this area.

2.5.1 Important Issues in the Modelling of Processes

Some of the factors that are important in the choice of model used are listed below.

- The properties of interest

  There are three main categories:

  Safety: Informally, a safety property is one which states that something bad will never happen, i.e. a specified property will never hold.

  Liveness: A liveness property states that something good will eventually happen, i.e. the program will eventually satisfy some specified goal.

  Fairness: A fairness property states that no process which is ready to run will be neglected forever, i.e. every process will get a 'fair go' in the long run. This is difficult to capture in a semantic model.

- The type of communication

  The early work on concurrent programming considered processes that communicated by accessing shared variables. More recent work mostly concentrates on processes that can communicate directly with one another. Communication can be modelled as synchronous (where the communicating processes must both be ready to communicate) or asynchronous (where one process can send a message even if the recipient is not ready).

- The type of nondeterminism

  A process may at some stage have a choice of possible computations to continue with. There are two possible interpretations of this. The choice could be made locally within the process or it may be necessary to refer to the global environment. Suppose, for example, a process has a choice of two actions A and B, and A needs to communicate with the environment. A local choice would enable the process to choose either A or B. Global choice would only allow A to be chosen if there was a communication partner for it, therefore avoiding the possibility of A waiting to communicate for ever.

This distinction is also known as angelic versus demonic nondeterminism. Angelic nondeterminism corresponds to local choices of the process or agent under consideration, while demonic nondeterminism corresponds to choices under the control of the external environment.

In an algebraic model of processes (e.g. Milner's CCS), the distinction between local and global choice corresponds to whether or not the sequencing operator is distributive over choice. If global choice is modelled then there is no distributive law, i.e. "a then (b or c)" does not equal "(a then b) or (a then c)".

Some models of concurrent processes have two distinct operators for local and global choice.

A linear model of time is adequate to represent and reason about local choice, but global choice requires branching time.

- Compositionality

  A concurrent program can be verified by constructing its semantic model and checking to see if it satisfies the program's specification. However, if the program is subsequently changed the whole process must be done again. This can make it very difficult and tedious to develop a correct program. Because of this, the emphasis in proof systems for concurrent processes has shifted towards verification as part of the design process (see [Homem & de Roever 86]). This requires that each process can be verified without reference to its external environment, and that the semantics can hide the internal activity of the process. This is possible with a *compositional semantics*, where the semantics of a program is a function of the semantics of its immediate syntactic subprograms.

- Representation of concurrency

  There are several ways to model the concurrent execution of processes:

  Interleaving The most common technique is to use nondeterministic interleaving of the actions from each participating process. This involves representing each possible computation of a process by a linear sequence of actions. A process may be nondeterministic, so it is represented by

the set of its possible computations. The behaviour of two concurrently executing processes is modelled by the set of all sequences that can be formed by choosing one possible sequence for each process and then interleaving their actions. If the two processes communicate synchronously, the corresponding communication actions from each process are treated as occurring simultaneously and are modelled by a single action. This can be extended to deal with any number of concurrent processes. The interleaving of two sets of sequences is represented by the operator '||' (shuffle) in formal language theory.

Both the linear and the branching temporal logics used in Computer Science model concurrency by interleaving.

Partial orders Another way of representing concurrent processes is to model the temporal precedence relation by a partial order on the set of action instances. (It must be action instances because an action can occur more than once). Potentially concurrent actions will not have any ordering between them.

Some examples of this are occurrence graphs (which can be generated by 'unfolding' a Petri net) and the *pomset* model of Pratt (1986). A pomset is a partially ordered multiset, which is a generalisation of a string. (A string can be viewed as a totally ordered multiset.)

Concurrency as a primitive operator It can be difficult to reason about concurrent processes using the methods above, especially for the interleaving model where the sets of sequences can be extremely large. An alternative method is to use one of these techniques as the underlying semantics for a formal system with an explicit concurrent composition operator. The semantic model can be used to construct inference rules for the new system. These rules can be used to reason about the behaviour of the program, and in some cases the lower level semantics can be forgotten completely (if the inference rules are complete).

Some examples of this are Milner's CCS [Milner 80] and Stuart's Regular

Propositional Temporal Logic (RPTL, see below) which has a complete axiomatisation.

2.5.2 Some Models of Processes used in Computer Science

This section gives a brief description of some of the methods used to represent and reason about concurrent processes in Computer Science.

Abstract programming languages

e.g. Calculus of Communicating Systems (CCS) [Milner 80] and Communicating Sequential Processes (CSP) [Hoare 85]

These languages are intended to highlight the important details of concurrent and communicating processes. Concurrency is specified using an explicit concurrent composition operator and communication between processes is usually modelled as occurring synchronously. Much of the work on the semantics of concurrent processes has concentrated on CCS and CSP.

CCS represents the behaviours of concurrent processes by terms consisting of atomic actions (which are communications with other processes) and operators on terms (e.g. concurrent composition, nondeterministic choice, etc.). Systems can be defined by recursive equations on terms. The behaviour of a system can be verified by expressing both the system and its specification as terms and using equational laws to show that the two terms are equivalent. The term representing the system will express more detail (e.g. showing how the system can be made from simpler components) and can be considered as the implementation of the specification. In general, however, the equational laws are not complete so the underlying semantics must be used as well. [Milner 80] presents an operational semantics based on a transition system (see below).

An important notion in CCS is that of observation equivalence. Two terms are said to be observation equivalent if their behaviours are indistinguishable to any external observer.

Petri nets (see Section 2.4.1)

Petri nets were developed as a tool for systems design and analysis. They model concurrency in a natural way — by causal independence. However, Petri nets are difficult to reason about formally, so they really need to be provided with their own semantics.

Operational semantics and transition systems

An operational semantics is one which defines the meaning of a language in terms of an interpreter (e.g. an abstract machine) which can be used to simulate the execution of a program, evaluate an expression, etc.

The most common type of operational semantics is a nondeterministic automaton called a transition system. The transitions describe how a process can move from one state to another, and are labelled to show how they synchronise with events in the environment. Concurrency is represented by interleaving. This does not result in a compositional semantics without additional complications.

Transition systems give a good intuitive model of concurrent processes and can be used to build a more abstract semantics.

Trace semantics

A trace is a finite sequence of communications that records a possible communication history of a process. A process is represented by the set of traces that it can produce. This can be generated from a transition system by finding the possible sequences of transitions for a given process and ignoring all internal actions. In order to use trace semantics in a compositional proof system, it is necessary to hide all internal communications when two processes are composed concurrently. This can be achieved if a trace is never referred to directly, but only by its projections onto the communication sets of other processes.

Trace semantics can express safety properties but not liveness. There is a generalisation of trace semantics called failure set semantics, which essentially identifies for each trace if a deadlock could arise. This allows liveness properties to be expressed.

Denotational semantics

In denotational semantics, processes are represented as mathematical objects, based on the notion that a process defines a state-transforming function [Stoy 77]. Concurrent processes are modelled as elements of a structure called a 'powerdomain'. These represent the possible executions of the process. An ordering can be defined on a powerdomain, and this allows the semantics of recursion to be defined using fixed point techniques. Denotational semantics are compositional, and this along with other desirable properties makes this type of semantics a very useful mathematical tool. However, denotational semantics do not give a very intuitive model of processes.

Program logics

A program logic consists of a formal language for expressing properties of a program and inference rules that can be used to determine the truth of expressions in the language. There are two types of program logic: *endogenous logics*, where the formulae refer to one particular program (e.g. temporal logics), and *exogenous logics*, which are designed for reasoning about arbitrary expressions in the programming language (e.g. Hoare logics — see below). The formulae of an exogenous logic mix logical assertions with program terms. This type of program logic can also be seen as providing a *logical semantics* for a programming language where the executions of a program must conform to the axioms and inference rules of the logic. The axioms determine the meaning of primitive constructs in the programming language, while the meanings of composite programs can be derived using the inference rules corresponding to the appropriate operators of the programming language.

In a *Hoare logic* the correctness formulae are of the form

$$\{p\}\ S\ \{q\}$$

where p and q are assertions about program states and S is a statement or expression in the programming language used to represent processes. This represents the statement that if p holds before S is executed, and S terminates, then q will hold afterwards. This is called partial correctness. A Hoare logic has rules of inference corresponding to

each operator of the programming language. These allow a correctness formula for the whole program to be derived from formulae concerning its syntactic subprograms.

For a concurrent programming language, the assertions p and q could be conditions on the values of variables (if processes communicate via shared variables), conditions on traces, or may refer to some other semantic model.

Hoare logic was introduced in [Hoare 69] for a simple sequential language, and many other researchers have investigated extensions for other types of language. [Loeckx & Sieber 87] includes a good overview of this type of logic.

Pratt (1976) introduced *dynamic logic* which considers programs as modal operators, so an expression of the form $\langle a \rangle p$ informally means "program a can terminate with p holding on termination". An associated operator can be defined as $[a]p = \neg\langle a \rangle \neg p$ and this informally means "whenever a terminates, p holds on termination". Thus a formula $\{p\}\ S\ \{q\}$ in a Hoare logic can be expressed in dynamic logic as $p \Rightarrow [S]q$.

2.5.3 Temporal Logics in Computer Science

In the specification and verification of concurrent programs, a temporal logic formula represents a set of possible execution sequences (in the case of linear time), or a set of trees (in the case of branching time). The initial work in this area concentrated on linear propositional temporal logics. However, this type of temporal logic is not expressive enough to represent some of the standard control structures of computer languages, e.g. loops, and also cannot express the existence of alternate computation paths. Because of this, computer scientists have developed a number of extensions to temporal logic that increase its expressiveness. Some of these are outlined below.

Extended Propositional Temporal Logic (EPTL) [Wolper 82]

This logic augments propositional temporal logic with temporal operators corresponding to right linear grammars. Such an operator represents all sequences that can be generated by the corresponding grammar. The resulting logic is expressively equivalent to temporal logic with quantification over propositions, and ω-regular expressions (i.e. regular expressions on infinite sequences).

Computation Tree Logic (CTL) [Emerson & Clarke 82]

CTL is a branching time logic that has modalities A (for all paths) or E (for some paths) followed by one of the linear time operators F (sometimes), G (always), X (next-time) or U (until—a binary operator). This logic can express the existence of alternate execution paths, but cannot express fairness constraints.

CTL* [Emerson & Halpern 84]

This is an extension of CTL where a universal or existential path quantifier can prefix an arbitrary combination of the linear time operators F, G, X, U, F^{∞} (infinitely often) and G^{∞} (almost everywhere, i.e. for all but a finite number of states in the path). $F^{\infty}p$ abbreviates GFp and $G^{\infty}p$ abbreviates $\neg F^{\infty}(\neg p)$. They do not add any expressive power to the logic.

CTL* is also known as full branching time logic. This is very expressive but it is probably too computationally expensive to use, being decidable in triple exponential time (i.e. in time of order $2^{2^{2^{cn}}}$) [Emerson & Sistla 85].

Fair Computation Tree Logic (FCTL) [Emerson & Lei 85]

An FCTL specification consists of a functional assertion of the program's behaviour along with an underlying fairness assumption. The functional assertion has a syntax similar to CTL with basic modalities corresponding to "for all fair paths", "for some fair paths" followed by an arbitrary combination of the linear time operators of CTL. The fairness assumption is specified by an arbitrary boolean combination of terms of the form $F^{\infty}p$ (infinitely often p) or $G^{\infty}p$ (almost everywhere p).

FCTL is intended to be used for the automatic verification of finite state concurrent programs. Such a program can be considered as a finite model for a propositional temporal logic, and a model checking algorithm can be used to check if a given specification holds in that model. Branching time logics have better computational complexity than linear logics for model checking, but cannot express fairness constraints as easily. FCTL allows reasoning under a wide range of fairness constraints without having to use full branching time logic (CTL*).

According to Emerson and Lei, this logic demonstrates that the basic question when

choosing a logic for model checking should not be "linear or branching?", but rather "which modalities do I need?".

Regular Propositional Temporal Logic (RPTL) [Stuart 86a]

RPTL extends propositional temporal logic by adding regular expression operators. This is expressively equivalent to EPTL for finite sequences, but RPTL cannot express infinite sequences. However, regular expressions are a more natural way of expressing sets of possible execution sequences and the logic requires fewer operators than EPTL. There is also a parallel (interleaving) operator which is included for convenience, although this does not increase the expressive power of the logic.

Stuart has used this logic in a multi-agent plan synchroniser, which inserts synchronisation primitives into a plan involving multiple agents to ensure that the specified safety constraints hold.

Regular Branching Logic (RBL) [Stuart 86b]

This is a branching time version of RPTL which can express the existence of alternate computation paths.

2.6 Formal Language Theory and Temporal Logics in Robotics and Planning

A number of researchers have used formal methods from computer science to represent robot plans and programs. This section briefly describes some of this work.

Robot Schemas

[Lyons 86] presents a computational model for robot programming, called Robot Schemas (RS). This is designed to reflect the important features of robot programming. These features are inherent parallelism, the need for formal verification, close linking of perception and action, and the need for parameterised prototypical actions. Lyons' computational model consists of concurrent computing agents that communicate through

input and output ports. Each agent is an instance of some generic agent called a Schema. These *schema instantiations* (SI's) can be dynamically created and linked to other SI's. A linked network of SI's can be treated as a single SI.

Lyons develops formal semantics for RS based on abstract machines called Port Automata. He also proposes a two step design methodology for developing programs using RS:

1. Top down development of the specifications for the schemas. These consist of temporal logic formulae and *schema abbreviations* (partial declarations of the parameters and connections of the schemas).

2. Bottom up coding of the schemas to meet their specifications.

Plan Synchronisation

Stuart (1985) presents a technique for resolving conflicts in multiple agent plans by generating safety constraints in a propositional temporal logic and then adding synchronisation primitives to the plan. These ensure that execution sequences disallowed by the safety constraints are not possible. This is based on the work of Wolper (1982) and Emerson and Clarke (1982) who have developed synchronisers for parallel programs.

This approach can be taken further by expressing the original plan in a temporal logic as well as the safety constraints. The generality of this technique depends on the expressiveness of the logic, and Stuart has investigated temporal logics incorporating regular expressions (RPTL, [Stuart 86a]) and branching time (RBL, [Stuart 86b]).

This technique may be useful in assembly planning where constraints on the action ordering could be generated by considering each object in turn and then be combined with extra safety constraints to produce an overall plan.

Task Grammar

Vijaykumar et al. (1987) use a representation based on context free grammars to analyse the possible behaviours that can result from a given task. Context-free grammars were chosen because "sequentiality is readily represented" and "the use of non-terminals

provides a representation of grouping and abstraction". References are given to two earlier uses of formal languages in robotics by Albus and Saridis.

2.7 Choosing a Temporal Representation Scheme

When choosing a representation of time it is important to understand the significant features of the problem that may influence the choice. Valdes-Perez (1986) presents some criteria to apply when analysing a particular representation scheme. This section lists some relevant questions from a slightly different perspective: that of analysing a problem to find an appropriate representation. This includes some points from Valdes-Perez.

The nature of the domain

- Should the model be point or interval based, or mixed?
- Should the model be based on linear or branching time?
  e.g. Are alternate action sequences needed?
- Is there inherent parallelism in the world being modelled? This could suggest that an explicit parallel composition operator would be useful.
- What is the nature of constraints that arise naturally in the domain?
  e.g. In assembly robotics, the constraints that arise from spatial considerations cannot in general be represented by a partial order. However, it may be possible to find a more powerful (but not complete) class of constraints that does cover most situations.

The nature of the problem

- What is the form of the required solution?
  e.g. In a planner, the solution is a sequence (or partial order) of actions that produces the desired result. For opportunistic scheduling, the on-line system must at any stage be able to determine efficiently which actions can occur next. These two types of problem suit different types of representation.

- Does any other information need to be extracted from the model?

  e.g. It may be necessary to access the constraints represented in the temporal model at run-time (for replanning, answering questions, etc.). Some representations (like a Petri net) may be useful for generating the desired behaviour, but may be difficult to analyse at run-time.

- Does the temporal model need to be updated incrementally?

  e.g. A problem involving search may require constraints to be added and/or subtracted as the search progresses.

- Does the problem have a natural hierarchical decomposition?

  If this is the case, a representation that can model the hierarchy may be useful.

Efficiency and complexity

There is a trade-off between the expressive power of a temporal representation and the computational cost of testing for consistency, etc. This makes the following questions important.

- Is completeness necessary?

  Do we need to generate all solutions to the problem?

  Do we need to represent constraints of arbitrary complexity?

  (Constraints that are too complex to be represented in the system could be incorporated by overconstraining the actions concerned. However, this could exclude possible solutions and may necessitate backtracking).

- How vital is it that all inconsistencies are detected?

- Is deduction well focused?

  i.e. Are many useless facts generated? This could be helped by 'clustering' related facts in the temporal model.

Convenience

- Is the temporal model a module in a larger system?
  If so, it may be necessary or desirable for the model to be in harmony with other data structures in the system.

- Does the temporal model support human interaction?
  If the system involves human interaction then there is an obvious advantage in using a representation that humans find 'natural'. Hopefully, this may allow better reasoning techniques and heuristics to be discovered, as well as being easier to think about.

Chapter 3

Behaviour-Based Programming

3.1 A New Methodology

For many years researchers have been trying to increase the flexibility and autonomy of mobile and assembly robots by applying the traditional techniques of AI. These methods can be characterised by their use of explicit symbolic representations of the world and its properties, and the central role played by these internal models in the computational processes of the agent. Typically, the incoming sensor data is processed to extract a high-level description of the current state of the world, and this is used to update the world model. For mobile robots, this internal representation is analysed to determine an appropriate next action; while for assembly robots, the repetitive nature and strict requirements of the task require that the symbolic representation of the world model be used to generate a reliable plan of action before execution begins. It is therefore essential that the true state of the world can be either modelled or predicted to a high degree of accuracy, and this places great demands on both the planning and execution components of the robot control system.

To date, this approach has had little success as systems built in this fashion have become bogged down in the seemingly endless quest to bridge the gap between the abstracted denotations of the symbols and the external reality of the world. Because these systems are too fragile to cope with unforeseen circumstances or even small differences between the true state of the world and the internal model, researchers have searched for ways to represent the world and to plan to greater levels of detail,

and have sought to provide their robots with more general problem-solving abilities. These efforts have increased the complexity of the systems, bringing new encounters with the problems of intractability and the theoretical difficulties of reasoning about action (e.g. the frame problem) and making deductions from incomplete knowledge (e.g. non-monotonic reasoning).

Recently, a number of researchers have begun to question this approach, not only as a methodology for building robotic systems for particular domains, but also as a long-term research strategy for the development of artificial intelligence. The criticisms are based upon the view that this *classical approach* is founded upon:

1. *The wrong decomposition*

   Traditional AI systems are structured around a decomposition of the system into modules performing the different information processing functions required by the agent. Typically, there might be modules devoted to sensory processing, world modelling, planning and reasoning about the world, task execution and motor control. This system architecture tends to enforce a sequential flow of information, with each module relying on the performance of its predecessor. This makes it very susceptible to any bottlenecks or weaknesses in the individual modules, and it is difficult to achieve timely and reliable behaviour in response to external stimuli.

   In practice, of course, this modular structure may be built on top of a low level reactive component which provides quick motor response to certain signals. However, the architecture still presupposes a particular relationship between sensing and action, with a general and integrated interpretation of the sensory data being used to control the actions of the robot.

   A radically different approach was proposed by Brooks (1986), who advocates a decomposition of the robot control system in terms of individual task-achieving units of behaviour. Each behaviour fulfils some informally specified imperative such as 'don't hit things' or 'follow moving objects' and contributes to the system's ability to perform reliably in an uncertain world. The overall competence of the system is developed incrementally, with a set of complementary behaviours being

added at each stage to provide the system with a limited but reliable and self-contained subset of its desired functionality. This forms a hierarchical structure of *levels of competence*, with the set of behaviours in each level extending and constraining the behaviours generated by the lower levels. For example, a typical hierarchy might have an initial layer of behaviours devoted to avoiding collisions with objects (both stationary and moving), followed by levels implementing a tendency to wander, more directed exploratory behaviour, map building, detection of changes in the environment, and so on up to higher level reasoning abilities. Within each level there is a tight coupling between sensing and action.

In the mobile robots developed by Brooks and his group at MIT [Brooks 90], the collection of behaviours within each level of competence is implemented by a number of simple computational modules that are specified as augmented finite state machines (i.e. FSMs with added registers and alarm clocks) and distributed across multiple microprocessors. These modules execute independently and in parallel, communicating asynchronously through hard-wired low bandwidth channels, with little or no central control or data structures. Output signals from new modules can be used to inhibit the inputs or suppress the outputs of existing modules in the same or lower levels, and this allows conflicts between actuator commands issued from different levels to be resolved and enables the system to be developed incrementally without modifying the existing parts. This is called the *subsumption architecture* [Brooks 85].

2. *The wrong methodology*

Traditionally, research in AI has concentrated on attempts to implement high level cognitive behaviour, inspired by our ability to reason about the world using symbolic representations and abstracted descriptions of objects and properties. It was generally assumed that once these problems were solved, it would be a relatively simple matter to equip the resulting symbol processing systems with sensors and actuators so that the symbols were manipulated in accordance with the real world behaviour of their denotations, for example:

> No model theory can specify what kinds of entity constitute the

> universes of its models. It refers only to the presence of functions and relations defined over a set, not to what it is a set of. And we could always make our universes out of entirely unsuitable things, in particular the tokens themselves.
>
> ... But I suggest that for the purposes of developing a naive physics, this whole issue can be safely ignored. We can take out a promissory loan on *real* meanings. One way or another, parts of our growing formalism will have to be attached to external worlds through senses or language or maybe some other way, and ghost models [i.e. Herbrand models] will be excluded. We must go ahead trying to formalize out intuitive world; paying attention indeed to the complexity and structural suitability of our models, but not worrying about what sort of stuff they are made from. [Hayes 85, pages 473, 474]

However, this presupposes that the researcher's intuition is correct about the abstractions that we use for reasoning, and that once the high level reasoning functions have been implemented, the symbolic representations of these abstractions can be successfully grounded in the real world via the sensory and motor control systems. For this to be feasible, there must exist *a priori* a task-independent level of representation of the world, which will provide the meeting point for the top-down refinement of the high level reasoning processes and the bottom-up development of the perception systems. There is no reason to think this is the case; in fact, there is evidence to suggest that perception processes are task dependent (e.g. [Yarbus 67], referenced by Brooks (1990)). Certainly, there has been little success to date in attempts to develop general purpose vision systems and task level robotic assembly languages.

An alternative viewpoint that is currently gaining much favour is that this process of abstracting the appropriate task-oriented behaviours and representations away from details of perception and actions is the hardest part of intelligence, and the correct solution can only be found by developing intelligent systems from the bottom up. Brooks (1986) argues that "mobility, acute vision and the ability to carry out survival related tasks in a dynamic environment provide a necessary basis for the development of true intelligence". In other words, once we can build systems with the ability to move around in an unsympathetic dynamic environment, sensing and reacting to a sufficient degree to ensure survival

and the competent performance of simple skills, we will then have the correct abstractions, and the higher level cognitive skills can be added with relative ease. This methodology therefore dictates that the appropriate starting point for research is to build artificial creatures that can perform simple tasks in a dynamic and unstructured environment. This is in contrast to the traditional approach which has concentrated on developing systems to perform complex cognitive tasks in simple and specially structured 'toy worlds'.

3. *Unnecessary Complexity*

   Another objection, related to the two previous points, is that the classical approach to robotics is based on principles that inherently require algorithms and systems of high complexity. First, the use of an explicit world model requires that all knowledge about the world should be amalgamated and converted to a symbolic form. Apart from the problem of choosing a common symbolic representation, and the simplification of sensor data that may be required to do this successfully, a central world model often encourages the use of needlessly complex and computationally expensive algorithms. Brooks follows the philosophy that "the world is its own best model" [Brooks 90] and has had remarkable success with complex systems — such as a robot that locates and collects empty drink cans [Connell 90] and a six-legged walking robot [Brooks 89] — that coordinate the different modes of their behaviour without direct communication between the relevant modules. Instead, the relevant behaviours are initiated by directly sensing in the world the effects of the activity or inactivity of other behaviours.

   A related approach is the use of non-symbolic representations that closely reflect the structure of the world being modelled. Using such an *analogical representation* may allow computations to be expressed much more naturally, simply and efficiently than a symbolic representation would allow. For instance, Steels (1988) has shown how the interactions of dynamic processes acting on a cellular grid can solve two dimensional path planning problems.

   Another source of complexity in the classical approach is its associated ideal of generality. Traditionally, the world models and reasoning processes have

been designed to be as general and task-independent as possible. An alternative view that is becoming increasingly popular is stated by Malcolm and Smithers as follows:

> It is a mistake to try to construct comprehensive world models suitable for the solution of all the kinds of problem with which the autonomous systems may be faced. To do so is to invite explosive complexity and consequent intractability. Rather . . . problems should be solved — in autonomous systems as well as in animals and people — in terms specifically and narrowly contrived for their solution, and which can be easily related to economically available subcognitive abilities. [Malcolm & Smithers 90]

Brooks (1990) argues that generality has become the default touchstone for success in AI systems as, traditionally, different aspects of intelligent behaviour have been investigated independently, without being grounded in the real world. As there is no way to test how well a single component will fulfil its role in a complete 'intelligent' system, its success must be measured by how well it performs on particularly difficult problems. With Brooks' methodology, the competence of the system to act in a complex environment is readily apparent, and it is counter-productive to add extra complexity simply to cater for situations that will rarely (or in fact, never) arise.

There are a number of other projects stemming from similar viewpoints, besides those discussed above. We outline some of them here to show the increasing interest in this emerging paradigm for robotics research.

A number of vision researchers are turning to the study of *active vision* (e.g. [Aloimonos, Weiss & Bandyopadhyay 87]) which is much more task-dependent than traditional approaches, and shows that the close coupling of sensing and action can greatly simplify the subsequent decoding of the sensory data.

In the areas of planning and mobile robotics there has been a lot of recent interest in the problem of representing control information for *reactive* or *situated* agents which have complex and conflicting goals or control rules that are responsible for generating competent and reliable behaviour in the intended domain [Rosenschein & Kaelbling 86, Kaelbling & Rosenschein 90, Georgeff & Lansky 86, Schoppers 87, Agre

& Chapman 87, Drummond 89]. The aim is to produce robust agents that can survive and perform tasks in the real world, while still having a notion of high-level goals and knowledge.

3.2 Behaviour-Based Programming in Robotic Assembly

The classical approach to assembly robotics can be seen as resulting from a particular computational model of the robot's role in the assembly process. Traditionally, the robot is treated as a peripheral output device attached to the computer with the sensors acting as input devices. Under this viewpoint, programming a robot becomes much like programming a computer, with the high level control constructs and procedure calls of the programming language being compiled down into primitive computations and communications with the input and output devices through the interfaces provided by the system. This encourages the programmer to structure the program in terms of separate sensing, reasoning and action modules with a central explicit world model — although this may be compiled away during the move from the off-line to the on-line system, or may in fact reside mainly in the programmer's mental model of the problem, and be only implicitly present in the program.

In the robot programming languages provided with most commercial robot systems, the interface for communicating with the robot is at the level of position control, i.e. the robot's actions must be defined in terms of a sequence of desired positions of the end effector. It is very difficult to program a robot reliably at this level of control because of the uncertainties of the real world: the limited accuracy of the robot and the sensors, the unavoidable variations in the dimensions and form of the parts, and the impossibility of predicting the effects of friction, etc. The programmer must therefore make appropriate use of sensors and clever assembly strategies to overcome these problems. However, with a position controlled robot and the computational model discussed above, there seems to be no principled way of deciding when or how to use the sensors to control the effects of uncertainty.

Traditionally, the basic form of an assembly program is designed as if for a robot operating in an ideal world, and sensing is added later on an *ad hoc* basis where it is

found to be necessary. Assembly planning research has followed a similar approach, with plans being generated for an ideal world and then refined or modified by further stages of planning or analysis. For instance, accurate part fitting can be achieved by planning and executing complex strategies for sliding the parts into place while in contact with other parts [Koutsou 86]. Another approach involves propagating expected or maximum errors throughout a sequence of actions or an assembly of parts in order to determine where the accumulated uncertainty could cause problems (e.g. [Brooks 82]). The plan can then be modified to reduce uncertainties at critical points by choosing a part of the robot's working area where the robot is more accurate for particular motions, or by introducing sensing actions. The addition of uncertainty analysis to the 'object-level' robot programming language RAPT [Popplestone, Ambler & Bellos 79] has also been investigated [Fleming 85]. However, techniques such as these are generally thought to be too computationally expensive and too limited for general use.

The solution to these problems is to build in the uncertainty reducing manipulation and sensing strategies beneath the level of the interface between the off-line programming language and the robot. By developing generic task-oriented *behavioural modules* which internally combine sensing with action as driven by the demands of the task, a 'virtual robot' is created, and this provides the appropriate level of physically grounded abstractions for reasoning about the robot's actions. In a similar fashion to Brooks' work, behavioural modules are designed and built to perform their task reliably across a range of situations, and are tested thoroughly in the real world. Under this methodology, solving an assembly problem is a two stage process. First, a suitable library of behavioural modules must be developed to provide the robot with the necessary abilities required for this class of problem. Once this is done, programming or planning can proceed in the usual top-down fashion. This is called *behaviour-based* assembly programming [Smithers & Malcolm 88, Malcolm & Smithers 90]. At Edinburgh, an ongoing research programme is investigating how these techniques can be extended. In particular, an important question is how individual behavioural modules can be combined, possibly with extra control constructs, sensing and actions, to form larger robust units of behaviour.

The application of these ideas to assembly robotics has a different flavour to their use in mobile robotics discussed above. In particular, Brooks goes to great lengths to avoid any form of symbolic representation and reasoning, and considers that "any particular planner is simply an abstraction barrier. Below that level we get the choice of whether to slot in another planner or to place a program which "*does the right thing*" [Brooks 87]. Brooks designs his systems to 'do the right thing' at all levels of abstraction, thus avoiding the complexity and fragility caused by generating and executing plans. However, the domain of assembly robotics has it own particular characteristics that make planning not only feasible but also desirable: a clearly defined goal, a predictable and well-engineered environment, and the commercial aims of repeatability, efficiency and predictability of execution. Also, the spatial and temporal constraints on the required robot actions mean that there is a strong strategic component to the assembly problem, and it is therefore beyond the capabilities of a purely reactive system.

Another distinctive aspect of assembly robotics is the importance of accurate fine motions which makes the management of uncertainty a key problem. To cope with this in an efficient manner, each behaviour should be designed to cope with some level of uncertainty, preventing its propagation, reducing it where possible, and removing any need for higher level processes to reason explicitly about these details. This type of encapsulation has been compared to the principles of VLSI circuit design [Smithers & Malcolm 88]. VLSI chips are constructed from predefined modules, each of which performs a particular logical function. These modules are designed to tolerate some level of error in the voltage and timing of input signals, and deliver their output with much less variation. New modules can be developed by combining appropriate existing modules. In the analogy with assembly programming, the modules correspond to task-achieving behaviours, each able to perform its task as long as the uncertainties in positions, orientations and dimensions of the objects involved are within a certain range. On completion of the task, the robot may know the object positions, etc., with more certainty. However, the problems of managing uncertainty in assembly robotics are much more difficult than this analogy suggests. It is not always possible to encapsulate uncertainty management strategies within individual behavioural modules.

The implications of choosing a particular strategy may affect all levels of planning. The important consideration, however, is to avoid *explicit* reasoning about uncertainty.

3.3 The SOMASS Assembly System

The SOMASS assembly system was developed to investigate and demonstrate the advantages of programming for robotic assembly in terms of task-achieving behavioural modules [Malcolm 87]. The key aim was to show that an appropriate behavioural decomposition of the assembly task provides a good basis to support and simplify the automatic planning of assembly tasks. It is a *hybrid* system, combining a symbolic planning component with a 'subcognitive' level of competent manipulatory skills implemented by behavioural modules. It is based on the principle that the validity of the behaviour-based approach can only be tested by building a complete working system without resort to simulations, synthetic data, etc. The system is designed for solving assembly problems in a simplified domain which highlights the fundamental problems of assembly, i.e. the shape dependent fitting of parts constructed with limited precision, whilst reducing or eliminating issues that are of less interest to this research programme (for the time being). The chosen domain is the assembly of objects using the seven pieces of the Soma puzzle.

The Soma puzzle, invented by the mathematician Piet Hein, consists of the seven distinct non-convex shapes (i.e. those containing a bend, protrusion or cavity) that can be formed by sticking together four or fewer equally sized cubes (Figure 3.1). The aim of the puzzle is to find a way of fitting the pieces together to form a 3×3 cube. Once that has been mastered, there are many other interesting structures that can be built from the parts, some of which are shown in Figure 3.2.

Although this is a fairly simple puzzle for humans, it has a number of features that make it a good experimental domain for assembly robotics research. First, the parts are geometrically simple and therefore easy for the robot to grasp and for the assembly planner to represent and reason about. This eliminates two problems of assembly that are orthogonal to the primary concerns of this project. The parts are also easy and cheap to construct, and this allows the generality of the assembly behaviours to be tested by

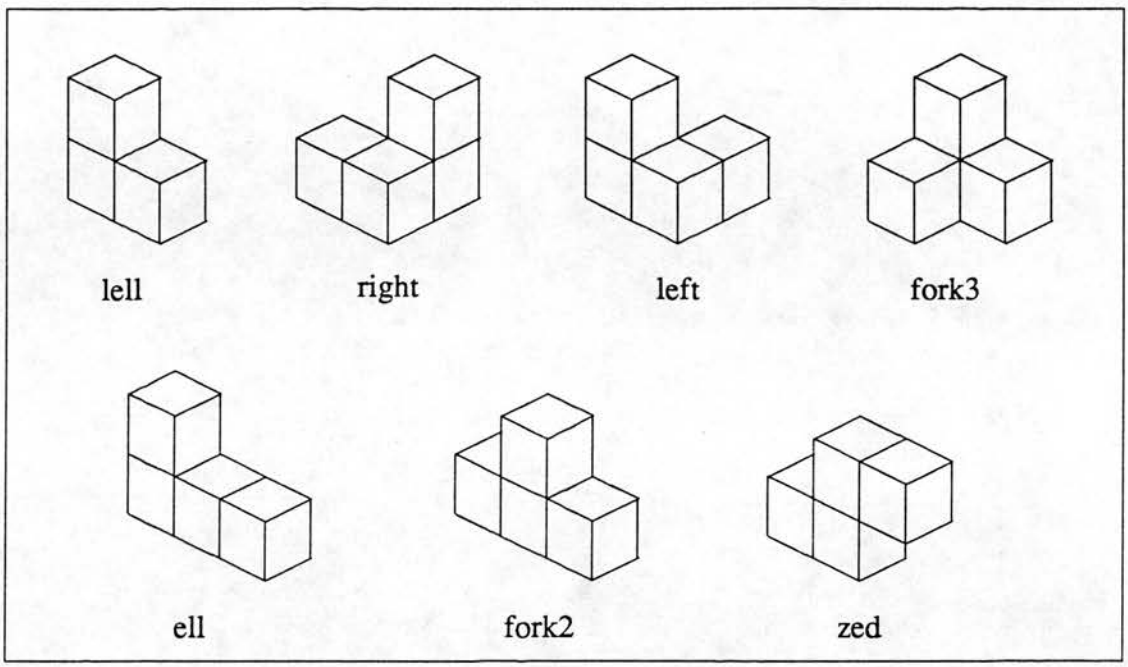


Figure 3.1: The Soma-4 pieces

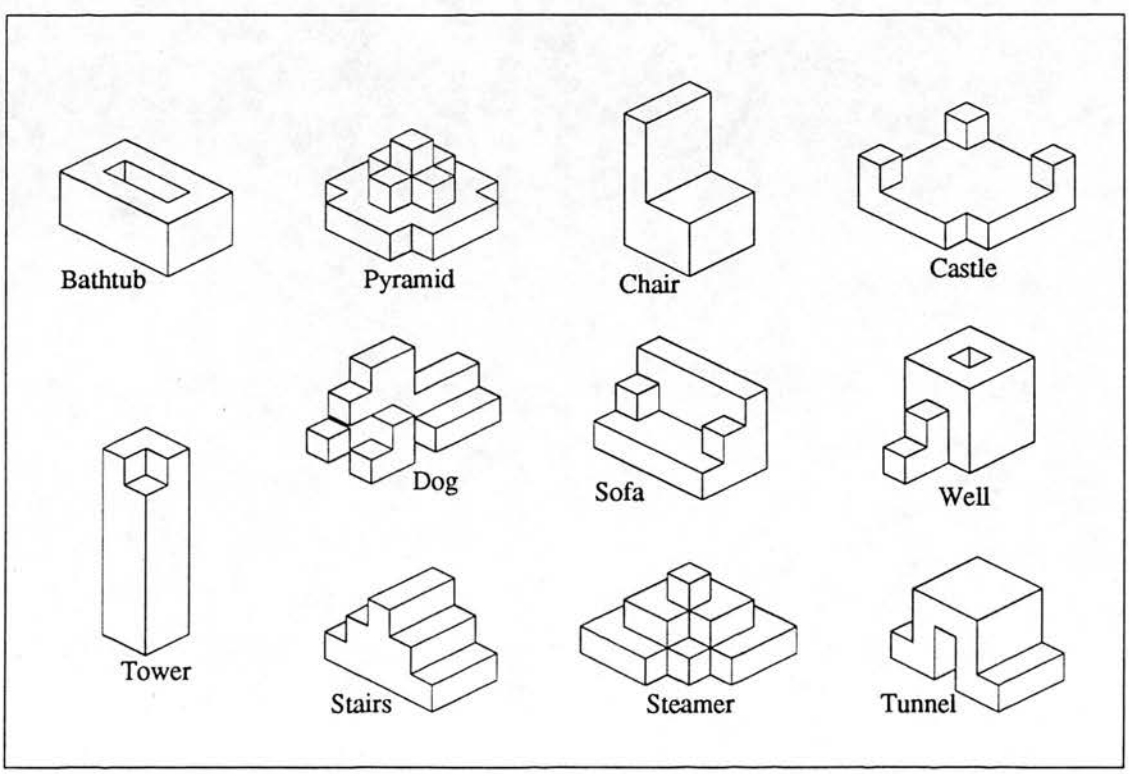


Figure 3.2: Some Soma-4 assemblies

using parts of various sizes and materials (although generally, wooden parts are used). This simplicity of structure does not, however, mean that the problems of uncertainty are reduced. The individual cubes from which the pieces are made (called *cubies*) are shaped with limited precision — there are errors of form of up to 5% of the basic cubie size — and these are then glued together roughly.

Second, by increasing the number of cubies that may be used to form a piece, the domain can be extended to include assembly problems of greater complexity. In the general case, we consider the set of non-convex shapes built from at most n cubies[1], and refer to the resulting assembly domain as the *Soma-n world*. When $n = 5$ there is a two-dimensional form of the 'peg in hole' problem; and the Soma-7 world introduces the full 3D version of this standard assembly problem. If we allow duplicates of the parts, then even the Soma-4 world contains problems with tightly fitting parts (due to cyclic adjacency relationships) that require the use of subassemblies.

The final advantage of this domain is the large number of different assembly problems that are possible. Not only are there many different shapes that can be built; there are also many different ways of building them. For instance, including reflections and rotations (because of gravity and the 'handedness' of the robot) there are 1440 ways of building a cube, approximately half of which are estimated to be physically possible to assemble.

3.3.1 Overview of the System

The system is based around a five degree of freedom Adept robot controlled using the robot programming language VAL2. In this language, robot motions must be described in terms of the destination position and orientation of the manipulator (absolute or relative to some coordinate frame) or by specifying the desired joint angles. In the current system[2], no sensors are used. However, the SOMASS system is designed to include sensing and work is now underway to develop part acquisition and placement behaviours that make use of touch sensors and visual feedback from an uncalibrated

[1] For economy, and to save the tropical rainforests, we may choose not to use *all* of these pieces!

[2] The SOMASS system is under constant development. In this thesis, the 'current' system refers to SOMASS 1.2, a stable version of the system described in [Malcolm & Smithers 90].

stereo vision system [Conkie & Chongstitvatana 90].

The system's aim is to plan and execute the assembly of the seven Soma pieces into a prespecified shape. As input it is given the approximate initial positions and orientations of the parts on the work-table, the length of the unit cube for the current set of parts (called a *cubit*), and a description of the desired final assembly shape. The parts are assumed to be presented in a standard configuration in which every part except the 'zed' part (see Figure 3.1) has a single uppermost cubie available for grasping. Each part is in a distinct location on the table, sufficiently far from other parts that it will not suffer any interference during the acquisition of neighbouring parts.

The planner attempts to find a suitable arrangement of the parts, a possible order of insertion, and pick-up and put-down grasps that allow the assembly to be constructed. If the two grasps are not the same, it must also plan a regrasping manoeuvre that allows the robot to change its grip. There are also some additional parameters that the planner must calculate. These are related to the strategies used to control uncertainty and will be discussed below. Note that the planner initially determines the position of the parts in the assembly. This is not normally part of the assembly process; however incorporating this ability into the planner allows it to generate its own test cases. The hierarchical nature of the planner makes it simple to bypass this stage of processing in order to plan the assembly of a particular predetermined solution to the Soma puzzle.

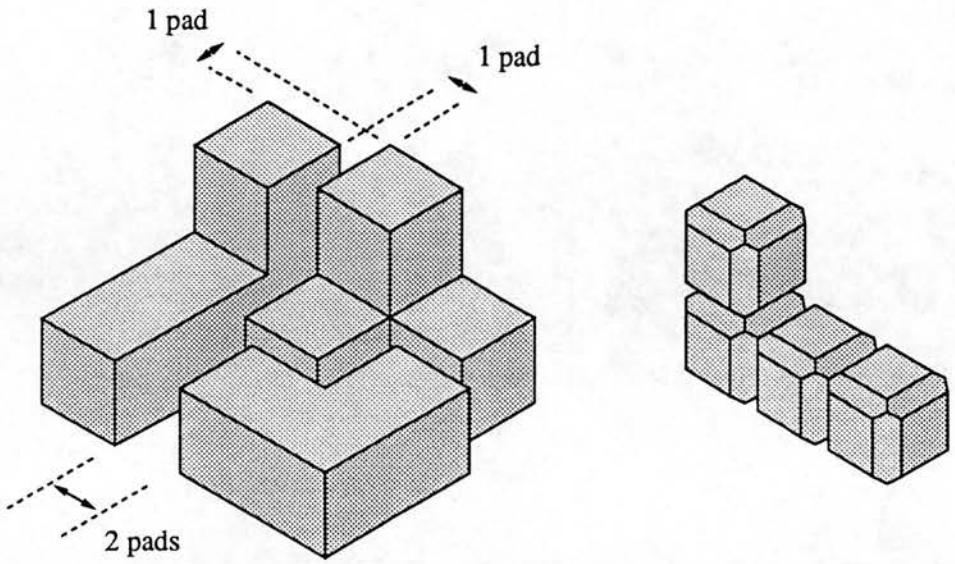
Once the plan is generated, the system executes it without human intervention. This is thought to be the first successfully implemented and fully automated assembly planning and execution system. The system is also very reliable — one plan has been executed 100 times consecutively with the robot running at twice the intended speed, without a single failure occurring. It has also been tested on 40 different plans for assembling the cube with part sets of two different sizes, and the resulting failure rate was less than 3%. Most of these failures were due to simple but previously undiscovered bugs in the assembly planner [Malcolm & Smithers 88].

Because of the lack of sensors, the pick-up behaviour in the current system uses a 'sweeping' strategy to reduce the initial positional uncertainty of the parts. Using a simple tool (the 'brush' — a block of wood of approximate dimensions $7 \times 1 \times 0.5$ cubits) the robot nudges the part from all four sides, thereby constraining its position

and orientation so that they are known to a much greater accuracy than the initial positional uncertainty allowed. The part can then be picked up, although for six of the seven parts the gripper reduces the uncertainty in position and orientation even further by giving an extra 'snap' on the uppermost cubie in the direction perpendicular to the final grasp orientation. This is not possible for the zed part.

In the case that a single grasp cannot be used for both picking up and placing a part, a regrasping strategy is used to change between the two grasps. The part is placed in a gravitationally stable position on a small raised platform called the *regrasp table*, and then picked up again using the new grasp. The raised regrasp table enables a part to be regrasped by a bottommost cubie, although the current system disallows this for the sake of reliability and because this restriction has little effect on the number of assemblies that can be built. No uncertainty managing strategy is used during the regrasp behaviour as the position of the part is known accurately enough to allow the regrasp to succeed.

The part placement (or *put-down*) behaviour is the most susceptible to failure due to uncertainty. This behaviour is responsible for inserting each part into the assemblage, and therefore the possibility of collisions with other parts arises. A small discrepancy between the expected and actual positions of a part could cause the manipulator or the part being inserted to knock another part with disastrous consequences. To avoid this, small spaces are left in the assemblage so that the parts are slightly offset from one another. This is referred to as *padding out* the assembly. The basic unit of padding space (called a *pad*) is determined by the put-down behaviour, but is generally about 1/8th of a cubit. However, the actual amount of space left between two parts may need to be larger than this due to the different length chains of adjacency that may exist between the two parts in the final assembly (see Figure 3.3a). Therefore, the planner calculates the x and y offsets for each part in terms of the number of pads by which it is displaced from its nominal position. Once all parts are in place, the assembly is gently 'patted' together from all directions using the brush, and this succeeds in 'squeezing out' any remaining positional uncertainty between the parts. This technique does not extend to the Soma-5 world where there may be cyclic adjacency relationships between two parts. Therefore, in more recent versions of the system, padding has been



(a) Padding out the assemblage (b) Using chamfered parts

Figure 3.3: Techniques for handling uncertainty in the assemblage

---

abandoned and the parts are made from chamfered cubies to facilitate their insertion into the assemblage (Figure 3.3b). An alternative approach is to pat the assemblage together at various times during the assembly operation.

Each behaviour leaves the gripper at a predetermined 'safe height' above the table whether it is holding a part or not. The next behaviour is responsible for moving the manipulator to the new desired position. It is guaranteed not to collide with anything as the safe height is chosen to give sufficient clearance above any possible assembly.

The three behaviours — pick-up, regrasp and put-down — are implemented using five behavioural modules which can be described as follows:

`pick(pos, grip)`

Move to position `pos` with the gripper oriented according to `grip`. Close the gripper and move upwards to the safe height.

The grip is described by a pair $\langle a_f, a_w \rangle$ where a_f is the axis (x, y or z) with which the fingers are aligned, and a_w specifies the alignment axis of the wrist.

`get(pos, grip)`

Move to position `pos` with the fingers aligned along the axis perpendicular (in

the x–y plane) to the one specified, 'snap' the gripper once, rotate to the correct orientation, close the gripper and move upwards to the safe height.

`pat(cpos,offset1,offset2,offset3,offset4)`

Pat around the boundary of the rectangle determined by the central position `cpos` and the offsets (in terms of cubits) in each direction.

This implements the sweeping behaviour that is used to reduce the initial uncertainty in the position and orientation of the part.

`put(cpos,x_offset,y_offset,z_offset, xpad,ypad,grip )`

Descend to the position determined by the parameters, open the gripper and move upwards.

This is intended for placing a part in the assemblage. `cpos` is the reference point of the assemblage, `x_offset`, etc. are offsets in terms of cubits describing the place in the assemblage in which the gripped cubie will be put, and `xpad` and `ypad` describe the required padding offsets in terms of pads.

```
manip(tpos,x_offset1,y_offset1,z_offset1,grip1
          x_offset2,y_offset2,z_offset2,grip2)
```

Descend to the position offset from `tpos` (the regrasp table reference point) by (`x_offset1`, `y_offset1`, `z_offset1`) cubits, with the gripper oriented according to `grip1`. Open the gripper, reorient according to `grip2` and then approach the position specified by `x_offset2` etc. along the axis of the wrist. Close the gripper and move upwards to a safe height.

This implements the regrasp behaviour.

For the purposes of this discussion, the names of the modules and the form of the parameters have been altered from those of the actual VAL2 procedures. In particular, we have given separate parameters for position and orientation. In the SOMASS on-line system these are represented as a single structure containing the x, y and z coordinates of the gripper as well as its roll, pitch and yaw. However, translating from one representation to the other is a trivial matter and the version given here helps to highlight the relationship between the behavioural modules and the representations used in the planner.

The pick-up behaviour is implemented by a call to `pick` to pick up the brush, followed by `pat`, and then `put` is used to replace the brush. The part is then picked up using either `get` (for all parts except zed) or `pick` (for zed). The regrasp behaviour is implemented by `manip`, and the put-down behaviour uses `put`.

For the assembly of the cube, SOMASS 1.2 executes an additional behavioural module once the assembly is complete. The module `patcube` implements the patting behaviour that is used to eliminate the padding spaces by gently pushing the assembly from all sides. This does not work for the other shapes and is therefore omitted from the plan when they are being assembled. This module has since been extended to cope with some (but not all) of the other shapes; however, the introduction of chamfered parts has now made this behaviour redundant.

3.3.2 The Assembly Planner

The SOMASS system assembly planner is very different in nature from those traditionally used in robotic assembly and domain-independent planning research. It is designed specifically for the Soma assembly task and therefore has the basic plan structure built in. The final plan will consist of a sequence of seven subplans describing for each part how it should be picked up, regrasped if necessary, and placed in the assemblage. There is no explicit representation of the operations that may be performed, or even those that have been selected as part of the plan. Instead, the plan is represented by a list of records, one for each part. The record for each part contains various values that are needed to fill in the behaviour parameters. These include the part's initial position, the pick-up and put-down grips to be used, and the final position. However, there is nothing that explicitly associates these values with particular behaviour parameters. The ordering of these records reflects the chosen order of assembly.

As a result the planner is very efficient. It is sufficiently specialised that operators are not needed. They have, in effect, been 'compiled away' — computations that would be expressed as operator selection and instantiation in a general purpose planner have been optimised and 'hard-wired' in. Instead of having a world model to represent the state of the work-cell at any given moment, it uses data structures that are well matched to the operations that must be done on them. The planner also takes advantage of

its specialisation by using failure-directed backtracking between the different stages of planning.

The planner is hierarchical, concentrating on particular aspects of the plan at each level. A complete plan is formed at each stage before it is elaborated at the next level. However, unlike traditional hierarchical planners, there is no explicit representation of abstracted operators, although its computations can be described in these terms. There are five levels, with chronological and (in some cases) dependency-directed backtracking being used to resume the search at a higher level whenever the search fails:

1. Find an arrangement of the parts within the assembly. This is done by a depth first search: for each part that remains to be fitted into the assemblage, choose a rotation and then a translation that allows the part to fit into the remaining gaps in the assemblage. This produces an abstracted plan corresponding to the use of a 'materialisation' operator in a perfect world, as if the (perfectly formed) parts could be made to materialise in the correct position in the assembly.

2. Find an order of assembly that allows each part to move vertically downwards without obstruction into a gravitationally stable position in the assemblage. This corresponds to an imaginary telekinesis operator which can magically move the parts from their initial positions, through the air and down into their final destinations in the assemblage.

3. Find a possible put-down grasp for each part. As each part is placed, the manipulator fingers and wrist must have sufficient clearance to avoid collisions with the other parts. A failure at this stage must be due to a previously placed part being in the way, and therefore the planner identifies the offending part and backtracks to level 2, skipping over all other plans which are identical up until the placing of that part (i.e. those that have the same initial segment). For some assemblies (e.g. the cube), it is never possible to insert the last part from above due to a lack of finger clearance. In this case, a special placement strategy is used: the final part is placed in a slightly offset position and is then pushed into place from the side.

4. Find a pick-up grasp for each part, with preference given to a grasp that matches the chosen put-down grasp. If this is not possible (which is usually the case), find a regrasping manoeuvre which allows the robot to change between the two grasps. This is the last stage of planning that can fail. The plan could now be successfully performed in a perfect world. However, in practice this plan will almost certainly fail due to errors in the form and the positions of the parts.

5. Determine the padding spaces that should be left between the parts to ensure that they can be inserted into the assembly (this technique is only applicable in the Soma-4 world).

There is an additional final stage of the planner which generates the parameterised VAL2 procedure calls from the planner's internal record structure. However, this is simply a straightforward translation process and is perhaps best considered as a preprocessed part of the execution control system.

Besides dependency-directed backtracking, the planner makes use of various other domain-specific techniques for optimising the search for a plan. Before the actual planning commences, the planner precomputes each distinct rotated form for each part together with all translations of this shape that could fit somewhere in the assembly. Note that different rotations of a part may lead to the same shape because of symmetries. Although these are not distinguished in the search for a general solution at level 1, they must be considered during regrasp planning. The list of rotations is also sorted to favour those configurations which have a single uppermost cubie available for grasping. The general solution finder also uses look-ahead pruning, analysing the remaining 'hole structure' in the assemblage to detect when the remaining parts will not fit. The Soma-4 set contains one part with three cubies and the rest all have four. By always choosing the position of the 3-cubie part first, the planner can reject any assemblage in which the unfilled cubie-sized spaces (called *cubicles*) are not partitioned into holes whose size is a multiple of four.

The planner is designed to have as little explicit knowledge about the world and the assembly domain as possible. Instead of performing complex reasoning about the world, it relies on the competence of the behavioural modules and the 'tacit knowledge' that is implicit in the assembly strategies used, the underlying assumptions about the

nature of the task and the work-cell environment, and a number of design principles embodied in the system. These principles take the form of restrictions imposed on the assembly process and work-cell layout to eliminate some possible sources of assembly failure, uncertainty, or system and computational complexity. Some of these restrictions are necessary, given the nature of the robot and the task. Others are introduced to increase the reliability or efficiency of the system. The design principles used in the SOMASS system include the following:

- It is assumed that all motions of the robot can be safely made at the prespecified safe height (related to the size of the parts) without any danger of collision.

- The parts may only be picked up from their initial positions by an uppermost cubie, with the gripper approaching from above. Similarly, the parts will only be placed into the assemblage from above with the wrist oriented downwards (with the possible exception of the last part to be placed). This design decision is also reflected in the test for gravitational stability of parts: although a part may be stable even if there are some unfilled cubicles underneath it, these could never be filled by inserting a part with a downwards motion. Therefore, a part is only considered stable if *all* spaces beneath it are occupied.

- When a part is inserted into the assemblage, no cubie may protrude above the destination position of the gripped cubie. This simplifies the problems of ensuring that the gripper and wrist have sufficient clearance.

Chapter 4

Planning with Behavioural Operators

One of the advantages of the behaviour-based approach to robotic assembly programming is the potential it gives us for applying traditional AI planning techniques to the assembly problem. In the classical approach, the complexity of assembly planning is greatly increased by the need to represent and reason about uncertainties in the position and dimension of the parts. In contrast, the SOMASS system shows that describing the robot's actions at an appropriate level — in terms of a repertoire of robust task-achieving behaviours — allows the planner to reason at a level much closer to the 'ideal world'. However, the planner in the SOMASS system is quite unlike traditional AI and assembly planners. Although it can intuitively be viewed as planning in terms of the behavioural modules and abstracted operators (e.g. the materialisation operator corresponding to level 1 of the planner), it is difficult to make this notion precise due to the planner's specialisation and implicit use of tacit knowledge about the domain and the system's uncertainty-handling strategies.

In order to understand the relationship between the symbolic and behaviour-based components of a hybrid system such as the SOMASS system, and the implications this has for planning in behaviour-based robotic assembly systems, we need to develop a framework for representing and reasoning about behaviours. In this chapter, we discuss this issue, showing how the Soma-world planner can be considered as an optimisation of a more general planner based on the familiar notions of operators and states, and suggest how the implicit assumptions and knowledge of the SOMASS system assembly strategies and design rules can be understood in this context. This motivates the

development of the logical framework for behaviour-based robotic assembly planning which is outlined in Section 4.2.

4.1 Representation and Domain Knowledge in the Soma-World Planner

The structure and operation of the Soma-world planner is strongly constrained by the nature of the task, and by the abilities and requirements of the *virtual* robot and work-cell for which the planner must plan. To determine the implications this has on the task of planning, and to facilitate the extension of the planner to other domains and problems, we need to gain a better understanding of its operation. In particular, the following questions arise:

- What does the planner represent and reason about, i.e. what entities of the world (both concrete and conceptual) and what aspects of these does the planner need to consider?
- What does the planner know about the behaviours?
- What domain and problem specific knowledge is built in, and how could this be expressed explicitly?
- How do the different stages of the planner relate to the different behaviours and their domains of competence?
  - Does each level correspond to a level of abstraction, as if it were planning for abstracted versions of the behaviours?
  - Can we show that a plan constructed at one level of the hierarchy is still valid when we move to the next, more detailed level?

4.1.1 The World Model

We begin by considering the world model used by the planner. There is no explicit notion of state, or representation of the current partially formed plan. Instead, all

information concerning the plan is kept in the assembly record structures. Once a general solution for fitting the parts together has been found, a set of records is created, recording for each part its initial and final configurations together with the possible transformations that map between the two. A part configuration is represented by a set of coordinate triples, representing the offsets in the x, y and z directions of the individual cubies with respect to the reference coordinate point for the part's current location. This location is the conceptual 'place' in the workspace where the part is currently located, and is not represented in the planner[1]. However, it is implicit that the initial configuration of a part is given relative to the reference point for its placement location, and the final configuration is given with respect to the reference point for the assemblage. The cubie offsets are given in terms of cubits, the nominal length of the side of a cubie.

The part transformations are given in terms of the possible rotations (given in a canonical form) that map between the part's initial and final orientations (taking symmetries of the part into consideration), together with a translation describing where the normalised 'shape' (i.e. set of cubie coordinates) of the rotated part fits into its chosen place in the assembly.

As planning progresses, the part records are ordered to reflect the selected order of assembly, and the 'get' and 'put' grips are generated and added to these records. If it is found to be necessary to regrasp a part, further information is added describing the part's normalised configuration on the regrasp table, the set of cubies on which it will rest while being regrasped, the two grips to be used (specified by the cubie to be grasped and the finger and wrist axes), and the rotations to be used when moving the part onto and off the regrasp table.

Finally, the required padding offsets for each part are determined and added to its record.

There is one other aspect of the world that is reasoned about during planning, although it is only needed in one level of the planner. This is the assemblage, i.e. the growing collection of parts which eventually becomes the completed assembly. It is

[1]This usage differs from that in [Malcolm 87], where 'location' refers to a combined position and orientation.

represented by two sets of triples giving the offsets from the assemblage reference point of the occupied and unoccupied cubies (respectively) within the outline of the final assembly shape. This is the closest that the planner comes to having any notion of state: during the search for a stable order of assembly, the planner simulates the successive states of the assemblage by updating the 'current' sets of occupied cubies and the holes left in the assemblage.

4.1.2 An Explicit Representation

The world model discussed above follows a minimalist approach: it is small, efficient and well-suited to the particular problem for which the SOMASS planner is designed. There is no central model of the world which is used; instead each level receives a record of the decisions made by the previous level, and also keeps its own private representations needed for its computations. Also, many details of the world that are needed for the final program are not needed and therefore not represented in the planner at all (not even symbolically), e.g. the positions of the parts, the regrasp table and the assemblage, and the current location of the robot end-effector. However, for a more general planner it is important that there is, at some level, a declarative description of the objects and their properties that are relevant to the planning problem, and the way in which the possible actions will affect these. Such representations make clear the correspondence between the inferences performed by the planner and the capabilities of the robot actions, and also allow the traditional techniques of AI planning to be applied where appropriate.

In order to describe the SOMASS planner in a more traditional operator-based framework, it is necessary to make explicit some of the aspects of the world that are currently left implicit in the way the planner operates. However, it is desirable to follow the principles of the SOMASS planner as much as possible, and in particular, to reduce complexity by leaving the planner ignorant of the details of the real world wherever possible. For instance, although the associated 'locations' should be explicitly represented alongside the sets of coordinates describing the configurations of the parts, the planner does not need to know where they lie in the workspace due to the design assumption that all locations are distinct and sufficiently separated to avoid interference

problems. We can therefore represent the distinct locations by symbolic names: `lell_loc`, ..., `zed_loc` for the initial locations of the parts; `regrasp_table_loc` for the location of the regrasp table; `assemblage_loc` for the place where the assembly is constructed; and `safeheight` to represent the volume of workspace above a certain predetermined height, which is guaranteed by the system design constraints to allow manipulator motions to be made without risk of collision. The behavioural modules move the end-effector to the appropriate part of the table using absolute movement commands, and therefore all positions in the `safeheight` zone are equivalent as far as the planner is concerned. `regrasp_table_loc` is included in the model for the sake of consistency; it is not really necessary as no behavioural module moves the regrasp table or leaves another object at that location. Every location (except `safeheight`) is assumed to have a local reference point so that the configurations of objects at that location can be defined in terms of offsets in the x, y and z directions. Furthermore, by convention (and the nature of the domain) the reference points are defined so that local offsets can be described as integer multiples of a cubit. The configurations of objects at the location `safeheight` are represented in their normal form.

We will also use symbolic names for the relevant objects in the Soma-world. For the Soma assembly problem there are ten objects to be modelled: the seven parts (`lell`, ..., `zed`; see Figure 3.1), the robot end-effector (`hand`), the regrasp table (`regrasp_table`), and the compound object formed by the growing conglomeration of parts being fitted together, i.e. the assemblage (`assemblage`). These have the types `part`, `end_effector`, `table`, and `assemblage` respectively. In the SOMASS planner, all the data-structures used to model the world can be considered to be associated with particular objects. In our operator-based description of the planner, we therefore define for each object a number of *attributes* of interest, whose values for any given state are assumed to represent all that the planner needs to know about the object. The defined attributes are shown in Figure 4.1.

Using assertions about the values of these attributes to represent sets of world states, the three robot behaviours can be modelled by *behavioural operators* whose effects and domains of competence are described using specifications such as those diagramatically depicted in Figure 4.2.

objects	attribute	value type
all objects	location	a location
lell, ..., zed	config	a set of cubie offsets
hand	holding	`nil`, or a pair (*object*, *grip*) (see below)
regrasp_table	contents	`nil`, or an object
assemblage	occupied_cubicles	a set of cubie offsets
	holes	a set of cubicle offsets

- *grip* is a term of the form

  `grip`(*gripped_cubie*, *wrist_orientation*, *finger_orientation*)

- *gripped_cubie* $\in$ `config`(*object*)
- *wrist_orientation* and *finger_orientation* are Prolog atoms (`x`, `y` or `z`), specifying their axes of alignment.

Figure 4.1: The attributes for the SOMASS system

The components of these operator descriptions are the name and formal parameters of the operator, the pre- and postcondition states, and a set of literals (i.e. atomic formulae or their negations) that constrain the terms that appear in the operator's parameters or in the pre- and postconditions. An operator specification states that whenever the operator is executed in a state satisfying the precondition, and all the literals are satisfied, the resulting state satisfies the postconditions. As with standard planning formalisms, a plan can be generated by searching for a sequence of operator instances that will achieve the desired goal state from the given initial state, with the added requirement that the literals associated with the chosen operators must all be satisfied during the course of planning.

The SOMASS planner can be viewed as a highly optimised and problem-specific form of a more generic planner that plans in terms of such behavioural operators, with the literals being evaluated as Prolog goals. The predicates appearing in these goals would be defined in an associated Prolog program describing the structure, physics and relationships of the Soma world assembly domain. The operator specifications given in Figure 4.2 are designed to highlight this viewpoint: the literals and their

pickup(Part, Location, GetGrip)

$$\left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{Location} \\ \text{config(Part)} = \text{Config1} \\ \text{holding(hand)} = \text{nil} \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{safeheight} \\ \text{config(Part)} = \text{Config2} \\ \text{holding(hand)} = (\text{Part, AfterGetGrip}) \end{array}\right\}$$

$$\begin{array}{l} \text{grip (PickupCubie, \_, \_)} = \text{GetGrip}, \\ \text{PickupCubie} \in \text{Config1}, \\ \text{possible\_get\_grip(GetGrip, Config1)}, \\ \text{normalised(Config1, GetGrip, Config2, AfterGetGrip)} \end{array}$$

place_in_assemblage(Part, PutGrip)

$$\left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{safeheight} \\ \text{location(assemblage)} = \text{assemblage\_loc} \\ \text{config(Part)} = \text{BeforePutConfig} \\ \text{holding(hand)} = (\text{Part, BeforePutGrip}) \\ \text{occupied\_cubicles(assemblage)} = \text{Occ1} \\ \text{holes(assemblage)} = \text{Holes1} \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{assemblage\_loc} \\ \text{location(assemblage)} = \text{assemblage\_loc} \\ \text{config(Part)} = \text{Config} \\ \text{holding(hand)} = \text{nil} \\ \text{occupied\_cubicles(assemblage)} = \text{Occ2} \\ \text{holes(assemblage)} = \text{Holes2} \end{array}\right\}$$

$$\begin{array}{l} \text{Config} \subseteq \text{Holes1}, \\ \text{fits(Part, Config)}, \\ \text{supported\_by(Config, Occ1)}, \\ \text{gripper\_clearance\_ok(PutGrip, Config, Occ1)}, \\ \text{finger\_clearance\_ok(PutGrip, Config, Occ1)}, \\ \text{config\_matching\_transformation(BeforePutConfig, Config, Trans, Rot)}, \\ \text{possible\_grip\_transformation(Trans, Rot, BeforePutGrip, PutGrip)}, \\ \text{Occ2} = \text{Occ1} \cup \text{Config}, \\ \text{Holes2} = \text{Holes1} \setminus \text{Config} \end{array}$$

Figure 4.2: Behavioural operators for the SOMASS system (continued on next page). See Appendix D for a description of the Prolog predicates used.

regrasp(Part, PutGrip, OnTabOffset, GetGrip)

$$\left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{safeheight} \\ \text{config(Part)} = \text{Config1} \\ \text{holding(hand)} = \text{(Part, Grip1)} \\ \text{contents(regrasp\_table)} = \text{nil} \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \text{location(hand)} = \text{safeheight} \\ \text{location(Part)} = \text{safeheight} \\ \text{config(Part)} = \text{Config2} \\ \text{holding(hand)} = \text{(Part, Grip2)} \\ \text{contents(regrasp\_table)} = \text{nil} \end{array}\right\}$$

possible_rotation(Grip2, Config2, GetGrip, R_Tab_Pos),
stable(R_Tab_Pos, R_Tab_Supported_Cubies),
translate_to_fit_table(R_Tab_Supported_Cubies, Trans, OnTabOffset),
translate_gripped_cubie(GetGrip, Trans, OnTabGetGrip),
translate(R_Tab_Pos, Trans, OnTabPos),
possible_rotation(Grip1, Config1, OnTabPutGrip, OnTabPos)

Figure 4.2 (continued).

---

ordering correspond closely to the computations performed in the SOMASS planner. For instance, the regrasp operator specification states that planning for the regrasping of a part takes place by first finding a solution for the most constrained part of the manoeuvre: the planner attempts to generate a possible rotation from the final put-down configuration and gripper orientation to a corresponding stable configuration on the regrasp table (this is the reverse of the action that will eventually be performed during the assembly). After normalising this configuration to fit on the table, and determining the regrasp table offset that corresponds to the gripped cubie in the pick-up grasp, a possible rotation is found to map between the initial pick-up configuration and the chosen configuration on the regrasp table.

There are, of course, many aspects of the SOMASS planner that are not easily expressed by considering the individual operator specifications in turn. During regrasp planning, for example, the pick-up grip is chosen before the rotation from the pick-up to the regrasp table configurations. This corresponds to evaluating the procedure call possible_get_grip(Grip1, Config1) immediately after satisfying the goal translate(R_Tab_Pos, Trans, OnTabPos). However, the call to possible_get_grip/2 is associated with the pick-up rather than the regrasp operator, and so this ordering cannot

be expressed in a system based on behavioural operator specifications alone. Additional facilities will need to be provided to allow goals to be ordered to achieve the maximum efficiency in the search. This would also allow the hierarchical structure of the SOMASS planner to be expressed, by delaying stability testing, etc. until a disposition of the parts within the assembly has been selected by evaluating the appropriate goals.

Other optimisations could be expressed by combining operator specifications using explicit ordering and control structures to provide declarative specifications for combined operators that encapsulate particular search strategies. For example, the SOMASS planner attempts to find a subplan for each part that does not require a regrasp operation, before it will try regrasp planning. These two types of plan require a slightly different specification for the place_in_assemblage operator. Forming compound operators for the two different types of subplan would solve this problem.

The discussion in this chapter suggests that a suitable basis for behaviour-based assembly planning could be achieved by developing a general logical framework based on the declarative specifications of behavioural operators, and which would offer the possibility of applying program transformation techniques to 'compile away' some of the explicit representations used, and to produce efficient partially evaluated specialised planners for particular families of assembly problems where the desired form of the plan is already partially determined (e.g. the Soma assembly problem).

For instance, a high level declarative description of the SOMASS planning problem could be optimised by abolishing the attribute `location` of `hand` as its value of `safeheight` is an invariant of all operators. Also, tests on the compatibility of the grips in the postconditions of pickup and the preconditions of place_in_assemblage (i.e. when `hand` is at `safeheight`) can be replaced by tests on the actual get and put grips, as in the SOMASS planner where the intermediate grips and configurations are not required to be represented.

4.1.3 Planning and Uncertainty

The behaviour-based approach to robot programming aims to deal with uncertainty within the behaviours, therefore hiding these details from the planner. However, there are two ways in which the SOMASS planner does seem to reason about uncertainty.

1. The planner must fill in (some of) the parameters for the VAL2 procedures that implement the Behaviours. This includes the sweeping procedure used to reduce the initial uncertainty in the position of the Soma pieces. This is a generic behavioural module that can be used for all seven pieces. However, the parameters that define the area of sweeping depend on the shape of the part being swept and therefore these must be calculated for each call of this module. For efficiency, this calculation should be done off-line, and indeed it is in the SOMASS planner (during the final translation to VAL2).

   In a more general and abstract planner, this sort of calculation should be conceptually separated from the other computations of the planner. The sweeping behaviour is designed to be always applicable when a part is required to be picked up from its initial position, and should succeed if the positional uncertainty is within the specified bounds. The decision to use sweeping was part of the process of designing a robust pick-up behaviour and does not need to be considered by the planner. The planner's responsibility is to ensure that this behaviour will only be applied in an appropriate context: the part being picked up must be in a known position and orientation (within the specified uncertainty bounds) and there must be no other object within a certain distance. This last criterion is guaranteed by only using a small number of distinct and separated locations.

   It seems appropriate to consider the calculation of the sweeping procedure's parameters as part of a generic behaviour which is parameterised only by the name of the part to be picked up, its location and a specification of the grip to be used. Part of this behaviour, however, is computed off-line. This suggests that the off-line system will consist of (at least) two stages: a planning stage, followed by a process of converting the behavioural operators into a form suitable for the on-line system.

2. The planner must also reason about the small spaces left between mating faces when placing each part in the assemblage. These 'pads' are intended to isolate the positional uncertainty created when placing each part, and prevent this from propagating throughout the assemblage. The completed assembly is patted

together afterwards.

The put-down behaviour is parameterised by the nominal positions of the parts together with the padding offsets in the x and y directions. However, the SOMASS planner only considers the nominal positions when checking the stability of the assemblage and ensuring that the robot hand has clearance for placing each part. In effect, it is working with idealised put-down operators that can place the parts in their nominal positions without problems caused by uncertainty. The padding spaces are added to the plan afterwards. For this reason the padding calculations were not included in the behavioural operator specifications given in Figure 4.2.

At first glance it might seem that the calculation of padding offsets could be thought of as an off-line component of the put-down behaviours, in a similar fashion to the treatment of the sweeping parameters discussed above. This could explain why the planner only considers nominal put-down positions in the main planning stage. There is a problem with this approach, however. In general, the declared effects of the behavioural operators should agree with the actual effects of the corresponding behavioural modules. In this case there is a discrepancy between the effects of the put-down operator and the corresponding behavioural modules which persists until the completed assembly is patted together in the final stage of the plan. The result of this is that the planner is incorrect when the dimensions of the Soma pieces are too small relative to the size of the unit padding space. When this happens, the stability of the padded assemblage no longer follows from the stability of the unpadded assemblage (which is what the planner checks) and a piece can fall off the assemblage. It would be very useful to have a framework for developing assembly plans where this sort of planning error could be discovered earlier!

One way of viewing the SOMASS planner which explains this problem is to regard the planner as constructing a plan using ideal put-down operators and then transforming it using the padding-patting assembly strategy. The transformation maps a plan using idealised operators into one that uses padded put-down operators, and will only preserve plan correctness if the stability and finger clearance conditions remain true. Therefore, although the idealised plan

may be correct for Soma parts of all sizes, when they become too small the transformation is incorrect because stability is not preserved. The effects of such a transformation may not only affect behaviours locally: the padding assembly strategy is spread across several behaviours, and therefore can affect behaviours occurring in-between.

4.2 A Logical Framework

The discussion in this chapter suggests the development of a logical framework for behaviour-based robotic assembly planning based on the combination of formulae describing the effects and competence of behavioural operators and compound plans. We will refer to these formulae as *plan specifications*. By describing the possible robot behaviours using plan specification formulae, and presenting a goal that specifies the desired plan (which may initially be either completely unknown or partially instantiated), the plan can be generated by proving the goal specification using the operator specifications as axioms. The inference rules will correspond to the ordering and control constructs that may be used to combine operators and plans to form larger compound plans. To enable the utility of various temporal languages to be investigated and compared, the framework will allow new temporal operators to be defined, provided that they have an appropriate compositional semantics so that corresponding inference rules can be derived. Flexible and powerful planning strategies will be able to be expressed and investigated using the technique of tactical theorem proving.

To motivate the description of the formal semantics given in the next chapter, this section presents a brief introduction to the world model used in our logical framework. The chapter concludes with a short example showing how a simple problem domain can be represented using this type of model.

4.2.1 The World Model

Motivated by the minimalist approach used in the SOMASS system planner, we have chosen to model the world as a set of *entities* that are relevant to the assembly task and a number of types to which these entities belong. The entities represent objects, locations

and other significant constituents of the world. Entities of each type have a number of *attributes* defined, and the values of these are considered to completely define the state of the world. The attribute values may be complex structured terms. This is a very simple but general representation and is based on the view that planning, in general, is an intractable problem unless the planner's view of the world is carefully structured to reflect the conceptual structure of the problem domain and the agent's competences. Until automatic problem-solvers can perform this structuring themselves, we must provide the appropriate abstractions. Note that the entity–attribute model encodes functional relationships, and although this requires special 'undefined' values to be used for partial functions, it helps to focus the planning process by encoding domain knowledge such as "every part must (normally) be *on* something"—the 'normally' refers to the case when the part is being held by the robot.

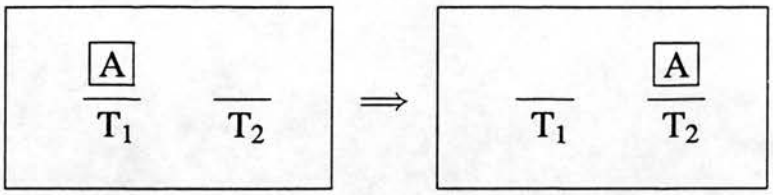
Example: A simple blocks world A simple blocks world can be modelled as consisting of two types of entities: *blocks* and *tables*. For simplicity, we assume that the tables are only large enough to support a single block and that blocks may not be stacked. Therefore the only type of problem we can express concerns the moving of blocks from one table to another.

The relevant knowledge about world states is encoded using the block attribute loc, recording the table on which the particular block is resting, and the table attribute on, which has value nil if the table is empty and otherwise records the block which is supported by that table. An atomic action for moving a block from one table to another can then be described by the following plan specification[2]:

$$\left\{\begin{array}{l}\mathrm{loc}(A) = T_1\\ \mathrm{on}(T_1) = A\\ \mathrm{on}(T_2) = \mathrm{nil}\end{array}\right\} \quad \mathrm{move}(A, T_1, T_2) \quad \left\{\begin{array}{l}\mathrm{loc}(A) = T_2\\ \mathrm{on}(T_1) = \mathrm{nil}\\ \mathrm{on}(T_2) = A\end{array}\right\}$$

[2]For examples we usually write a plan specification $\langle \phi, P, \psi \rangle$ simply as $\phi\ P\ \psi$

which corresponds to the following change in world state:



Initial State Final State

To plan in this blocks world this plan specification formula will be used as an axiom, and planning will take place by decomposing a goal formula specifying the desired plan using an inference rule for the sequential composition operator ';', until every subgoal is an instance of this axiom and can therefore be resolved away leaving the final plan as a sequence of 'move' actions.

In Section 5.3.3 this process is demonstrated for a simple block-swapping problem.

■

Chapter 5

A Plan Specification Logic

In Chapter 4, it was shown how the capabilities of task-achieving behavioural modules can be expressed using plan specifications that describe the behavioural modules' effects on a simple world model, and how Prolog goals can be used for complex pre- and postconditions. This chapter presents the formal description of a logic of plan specification formulae, and shows how planning can be expressed as inference in this logic.

To help the reader refer back to the definitions for the notation used in this chapter, these are indexed in a glossary of symbols that appears at the end of this thesis.

5.1 The Formal Model

A planning problem is modelled by a tuple

$$\langle \mathcal{L}, \mathcal{E}, \mathcal{A}, O, \mathcal{T}, Ax, G, \mathbf{C} \rangle$$

where:

- $\mathcal{L} = \mathcal{L}(\mathbf{F}, \mathbf{P}, \mathbf{X})$ is a first order language over a set of function symbols **F** (including constants), a set of predicate symbols **P** and a set of variable symbols **X**. The terms built using function symbols from **F** form the object-level universe, i.e. the ground data-structures used to reason about the world.

  Let the set of terms generated by **F** and **X** be denoted by $T_{\mathbf{F}}(\mathbf{X})$.

- $\mathcal{E}$ is a family $\{E_\tau \mid \tau \in T\}$ of disjoint sets of constant symbols from **F**. These represent the real-world entities that are of interest in the planning problem. T is a set of constants from **F** representing the different types of entity.

  The set of all entities is $E = \bigcup_{\tau \in T} E_\tau$.

  Let $V_\mathcal{E} = \{V_\tau \mid \tau \in T\}$ be a family of sets of variables from **X**. These are treated as typed entity variables, with the variables in each V_τ ranging over the corresponding set of entity constants E_τ.

- $\mathcal{A}$ is a family $\{A_\tau \mid \tau \in T\}$ of sets of attribute names. These are the names of the attributes associated with each type. The sets A_τ are not necessarily disjoint as an attribute name may be used for more than one type of entity.

- O is a set of function symbols. These correspond to the operators of classical planning and represent parameterised atomic actions. The set of operators of arity n is denoted by O_n.

  e.g. If *pickup* is an operator parameterised by a part name, a location and a grasp position, then $pickup \in O_3$.

  Atomic actions are terms of the form $a(t_1, t_2, \ldots, t_n)$ where $a \in O_n$ and $t_1, t_2, \ldots, t_n \in T_\mathbf{F}(\mathbf{X})$.

  The set of atomic actions is denoted by $\mathcal{P}_0$.

- $\mathcal{T}$ is a temporal language defined by a pair $\langle \Omega, \mathcal{I} \rangle$.

  Ω is a family $\{\Omega_n \mid n \in \mathbb{N} \cup 0\}$ of sets of function symbols, indexed by arity. These are the temporal operators used to construct plans from subplans. Temporal operators of zero arity will be special operators such as a 'do-nothing' action which take no plan arguments.

  The set of plans, $\mathcal{P}$ is defined recursively by:

  (1) All atomic actions are in $\mathcal{P}$

  (2) $\omega \in \Omega_n \wedge p_1, \ldots, p_n \in \mathcal{P} \Rightarrow \omega(p_1, \ldots, p_n) \in \mathcal{P}$

  e.g. if P_1 and P_2 are plans and ';' is a sequential plan composition operator from Ω_2, then $P_1; P_2$ is a plan.

Plans will be interpreted as sets of finite sequences of atomic actions. These are the possible execution sequences of the plan. Plans are interpreted compositionally, i.e. the set of possible execution sequences of a compound plan $\omega(p_1, p_2, \ldots, p_n)$ depends on the temporal operator ω and the possible sequences of the subplans $p_1, p_2, \ldots, p_n$. This is done via a set of functions $\mathcal{I} = \{\mathcal{I}_\omega \mid \omega \in \bigcup_n \Omega_n\}$ which interprets each temporal operator ω of arity n as a function $\mathcal{I}_\omega : \wp(\mathcal{P}_0{}^*)^n \rightarrow \wp(\mathcal{P}_0{}^*)$, where $\wp$ denotes the power set of a set X (i.e. the set of all subsets of X) and X^* denotes the set of all finite sequences of elements from X. For example, the sequence operator ';' can be defined as follows:

Definition 5.1 *The sequence operator* ';' $\in \Omega_2$ *is defined by the interpretation function*

$$\mathcal{I}_{(;)}(S_1, S_2) = \{\bar{p} \mid \bar{p} = \bar{p}_1\bar{p}_2,\ \bar{p}_1 \in S_1,\ \bar{p}_2 \in S_2\}$$ ∎

i.e. a possible execution sequence for a plan '$P_1; P_2$' must be formed from a possible sequence for P_1 followed by a possible sequence for P_2.

Given the interpretation functions for the temporal operators, the interpretation of plans is given by the function $I: \mathcal{P} \rightarrow \wp(\mathcal{P}_0{}^*)$, defined recursively as follows:

$$I(a(t_1, t_2, \ldots, t_m)) = \{a(t_1, t_2, \ldots, t_m)\} \qquad \text{if } a \in O_m$$
$$I(\omega(p_1, p_2, \ldots, p_n)) = \mathcal{I}_\omega(I(p_1), I(p_2), \ldots, I(p_n)) \qquad \text{if } \omega \in \Omega_n$$

- Ax is a set of axioms defining the atomic actions in O. These are the plan specification formulae discussed in Section 4.2, and will be formally defined in Section 5.1.1.

- G is a plan specification schema, i.e. a plan specification formula where the plan term is a variable or a compound term which may contain plan variables. This is the goal plan specification which describes the initial and goal states and may also partially specify the structure of the desired plan.

- **C** is a set of Horn clauses in the language $\mathcal{L}$. This is a Prolog program defining the predicates appearing in the plan specifications in Ax and G. **C** must include

a unit clause $entity\_type(e, \tau)$ for each entity constant $e \in E_\tau$. This is needed to express the typing of entity constants as $\mathcal{L}$ is an untyped language.

5.1.1 Plan Specifications

The aim of a planning system is to find a way of achieving a goal state starting from a given initial state, by the execution of actions under appropriate ordering constraints. To do this, the planner must have a representation of the effects of actions and the conditions that must be true for the action to have its intended effect. This information is represented by the axiom set Ax. This is a set containing a logical formula for each operator in Ω, asserting that if certain preconditions hold in a state, the new state resulting from executing an instance of that generic action will satisfy the given postconditions.

This type of formula can also be used to describe the effects of executing a plan, and in particular, to specify the plan that is to be generated — we want to find a plan P that will achieve the desired goal when executed in a state satisfying the relevant initial conditions. Hence, such a formula is called a *plan specification* (abbreviated *PlanSpec*).

A plan specification is an expression of the form:

$$\Gamma \triangleright \langle \phi, P, \psi \rangle$$

where

- Γ is a set of literals in the language $\mathcal{L}$, i.e. atomic formulae $p(t_1, \ldots, t_n)$ or their negations, where p is a predicate and $t_1, \ldots, t_n$ are terms from $\mathrm{T_F}(\mathbf{X})$. These are Prolog goals and are used to test complex preconditions and to construct terms that appear in the postcondition state specifications.

  Following the normal convention, we will usually treat sets of literals as if they are lists, writing "Γ, Δ" to denote $\Gamma \cup \Delta$ and "Γ, A" for $\Gamma \cup \{A\}$.

  If Γ is empty we simply write the plan specification as $\langle \phi, P, \psi \rangle$.

- P is a plan term, i.e. a term from $\mathcal{P}$.

- ϕ and ψ are sets of triples of the form (e, a, v) which represent the condition that attribute a of entity e has value v. An entity term e may be a constant from

E or a variable from $V_{\mathcal{E}}$. If the entity symbol e has type τ then a must be an attribute from A_τ. The value v is a term from $T_{\mathbf{F}}(\mathbf{X})$, i.e. a Prolog term. We will sometimes write triples (e, a, v) as equations $a(e) = v$ to make the intended meaning clear.

These sets of triples are called *state specifications* (abbreviated *StateSpec*) as they partially describe a state. If ρ is a state specification, let $|\rho|$ denote the set of syntactically distinct entity symbols appearing as the first element of some triple in ρ:

$$|\rho| = \{e \mid \exists\, (e, a, v) \in \rho\}$$

A state specification ρ is said to be *consistent* if it contains at most one triple (e, a, v) for each entity e and attribute a. A plan specification is consistent if both of its *StateSpecs* are. Despite its name, consistency of state specifications is a syntactic notion. However, under the definitions and semantics presented in the next section, a consistent state specification ϕ is always satisfiable (by some *state*) if certain conditions designed to reflect our intuitive notion of the meaning of a state specification are placed on the interpretation of the function symbols and variables occurring in ϕ.

From now on we assume that all *PlanSpecs* are consistent. We also make the reasonable assumption that no entity constant or variable will appear in the plan term or the *StateSpec* value terms of a plan specification without also appearing as the first element of some triple in the pre- and postcondition *StateSpecs*. This simplifies some of the technical details and ensures that an implementation can easily determine the set of entity symbols that appear in a *PlanSpec*.

Note that the 'syntax' of plan specifications described above does not define the structure of *PlanSpecs* using the standard notion of terms as trees or strings of symbols. In particular, state specifications are defined to be *sets* of triples, which cannot be represented by any term algebra. This type of generalised syntax is not uncommon in logical systems whose formulae are to be manipulated by machine; for example, the interactive theorem prover LCF [Gordon, Milner & Wadsworth 79, Paulson 87a] uses a sort of sequent calculus in which a sequent $\Gamma \vdash A$ is true if the assumptions in the

set Γ can be used to prove A. LCF operates directly on these sets, constructing the hypotheses for the conclusion of an inference rule to be the union of the hypothesis sets in the premises [Paulson 86].

5.1.2 Semantics of Plan Specifications

When planning, we want to regard plans as functions that transform world states in a predictable way if certain conditions hold. However, the interpretation I defined above models a plan P by its set of possible execution sequences, and there is no guarantee that every sequence of actions in $I(P)$ has the same effect on a given state. A plan specification asserts that every execution path of a plan achieves the same result, provided the plan's preconditions hold in the initial state. Unlike traditional logical models for planning, a *PlanSpec* completely determines the state resulting from the execution of the plan or action without requiring extra 'frame axioms'. These are axioms stating for each type of action which properties of the world are unaffected by it and will therefore continue to hold after the action is performed (see, e.g., [Georgeff 87, Genesereth & Nilsson 87]). The semantics for *PlanSpecs* has a form of frame axiom built in, and so all the information about the resulting state is included in the one formula. Therefore, we can formulate planning as a theorem-proving problem in a special-purpose logic of plan specification formulae. This will be discussed in Section 5.3.

A plan specification formula contains function and predicate symbols and variables from $\mathcal{L}$ as well as atomic action operators from O. The truth or falsity of a *PlanSpec* must therefore be defined relative to the meanings we attach to these symbols. This is done by defining the interpretations of the function, predicate and atomic action operator symbols, as well as the values assigned to the variables.

An *interpretation* for a planning problem is a pair $\mathfrak{M} = \langle \mathcal{M}, \mathcal{F} \rangle$ where:

1. $\mathcal{M}$ is an interpretation for the first order language $\mathcal{L}$, in the usual model-theoretic semantics for predicate logic, i.e. $\mathcal{M}$ consists of a domain of interpretation (denoted $|\mathcal{M}|$) for terms in $T_{\mathbf{F}}(\mathbf{X})$, and a function interpreting the function and predicate symbols in $\mathcal{L}$ as functions and relations over $|\mathcal{M}|$ (see, e.g.

[Enderton 72]).

For a predicate or function symbol s in $\mathcal{L}$, let $s^{\mathcal{M}}$ denote the interpretation of s under $\mathcal{M}$, and we extend this notation to sets: if $S = \{s_i \mid i \in I\}$ for some index set I, then $S^{\mathcal{M}} = \{s_i^{\mathcal{M}} \mid i \in I\}$.

Given $\mathcal{M}$, we define the set of possible states, $\Sigma_{\mathcal{M}}$, to be the set of all possible assignments of objects in $|\mathcal{M}|$ to the attributes of each entity. Formally, a state σ is a function mapping each attribute name $a \in A_\tau$ for some type τ to a function $a^\sigma : {E_\tau}^{\mathcal{M}} \rightarrow |\mathcal{M}|$.

We omit the subscript and simply write Σ when the interpretation $\mathcal{M}$ is clear from context.

2. $\mathcal{F}$ is an interpretation function mapping each n-ary action operator a to a function $f_a : |\mathcal{M}|^n \times \Sigma \rightarrow \Sigma$.

   If $p \equiv a(t_1, \ldots, t_n)$ is an atomic action and $\hat{t}_1, \ldots, \hat{t}_n$ are the interpretations of $t_1, \ldots, t_n$ in $|\mathcal{M}|$ (see below), then let f_p denote the state-transforming function $f_a(\hat{t}_1, \ldots, \hat{t}_n) : \Sigma \rightarrow \Sigma$.

   $\mathcal{F}$ extends to an interpretation function $\mathcal{F}^*$ on finite sequences of atomic actions. $\mathcal{F}^*$ maps the sequence $\bar{p} \equiv \langle p_1, \ldots, p_n \rangle$ to the function $f_{\bar{p}} = f_{p_n} \circ f_{p_{n-1}} \circ \cdots \circ f_{p_1}$, where $f \circ g$ denotes function composition: $(f \circ g)(x) = f(g(x))$.

A *valuation* $\mathcal{V}$ is a function mapping variables of the language $\mathcal{L}$ to objects in $|\mathcal{M}|$. If $t \in T_{\mathbf{F}}(\mathbf{X})$, let $t_{\mathcal{V}}^{\mathcal{M}}$ denote the interpretation of t under $\mathcal{M}$ and $\mathcal{V}$:

$$t_{\mathcal{V}}^{\mathcal{M}} = \begin{cases} \mathcal{V}(t) & \text{if } t \text{ is a variable} \\ t^{\mathcal{M}} & \text{otherwise} \end{cases}$$

and we extend this notation to state specifications: $\phi_{\mathcal{V}}^{\mathcal{M}} = \{(e_{\mathcal{V}}^{\mathcal{M}}, a, v_{\mathcal{V}}^{\mathcal{M}}) \mid (e, a, v) \in \phi\}$. Note that the attribute name a is unaffected as it is not part of the language $\mathcal{L}$.

We write $\hat{t}$ to denote an interpretation $t_{\mathcal{V}}^{\mathcal{M}}$ of t under some unspecified $\mathcal{M}$ and $\mathcal{V}$, or when $\mathcal{M}$ and $\mathcal{V}$ are clear from context.

Given an interpretation $\mathfrak{M} = \langle \mathcal{M}, \mathcal{F} \rangle$ and valuation $\mathcal{V}$, we can define a relative notion of truth called *satisfaction*. A literal $A \equiv \ulcorner p(t_1, \ldots, t_n) \urcorner$ from the language $\mathcal{L}$ is

said to be satisfied by $\mathcal{M}$ and $\mathcal{V}$, written $\models_{\mathcal{M}} A[\mathcal{V}]$, if $p^{\mathcal{M}}(\hat{t}_1, \ldots, \hat{t}_n)$ holds. A set of literals Γ is satisfied by $\mathcal{M}$ and $\mathcal{V}$ if each literal in Γ is satisfied by $\mathcal{M}$ and $\mathcal{V}$, and we write $\models_{\mathcal{M}} \Gamma[\mathcal{V}]$.

Informally, our intended notion of satisfaction for *PlanSpec* formulae is that $\mathfrak{M}$ and $\mathcal{V}$ will satisfy a *PlanSpec* formula $\Theta \equiv \ulcorner \Gamma \triangleright \langle \phi, P, \psi \rangle \urcorner$ if the following condition holds:

> *If $\mathcal{M}$ and $\mathcal{V}$ satisfy all the goals in Γ then the interpretations under $\mathcal{F}^*$ of all possible execution sequences for the plan P will have the same effect on all states satisfying the precondition ϕ. This effect will be to change the attribute values that are assigned new values in ψ, leaving all other attributes unaffected.*

To clarify the meaning of a *PlanSpec* let us consider the process of encoding a planning problem using the logical framework described in this chapter. The user has an informal notion of the relevant aspects of the world, and must express these in a logical form. The language $\mathcal{L}$ is used to describe the 'fixed' part of the world: 'objects' (e.g. blocks, colours, integers) are represented by ground terms from $T_{\mathbf{F}}(\mathbf{X})$, and fundamental relationships between objects are expressed as predicates from **P** which are defined by the clauses in **C**. The objects with properties that may be changed by the execution of a plan are declared using the entity and attribute model discussed above. It remains to describe how the atomic actions affect the world. This is done by giving a plan specification axiom for each action operator, and these serve to restrict the possible interpretation functions for these actions, just as the clauses **C** define the class of models for $\mathcal{L}$ that agree with our intuitive model of the world. A *PlanSpec* axiom declares that the corresponding action is generic[†] and predictable: the interpretation functions for all instances of that action have the same effect on all states that satisfy the preconditions.

Note that a *PlanSpec* is true if its precondition *StateSpec* is not satisfied — a *PlanSpec* says nothing about the plan's performance in states that aren't within its domain of competence. Similarly, if the Prolog goals in a *StateSpec* are not satisfied, the plan specification is vacuously true. These goals therefore limit the interpretations $\mathcal{M}$ and valuations $\mathcal{V}$ for which the *PlanSpec* conveys any information.

There are two additional restrictions on the intended interpretation of the constants and variables occurring in a *PlanSpec* Θ, both involving the interpretation of the entity

[†] i.e. having the same effect for all parameter instantiations.

symbols that appear in Θ.

1. The types of entity symbols must be respected. If v is an entity variable from V_τ then the valuation should map v to the image under $\mathcal{M}$ of one of the entity constants of type τ, i.e. $\mathcal{V}(v) \in E_\tau{}^{\mathcal{M}}$. In particular, there must exist an entity constant e in $\mathcal{L}$ of the appropriate type (expressed in **C** by the clause $entity\_type(e, \tau)$). Note that in the formal model, the type of an entity variable is determined by the partitioning of $V_\mathcal{E}$ by type. In an implementation the type would be given by the user as part of the term representing that variable.

2. The distinct entity symbols appearing as the first components of the triples in the two *StateSpecs* of a *PlanSpec* must map to different elements of $|\mathcal{M}|$, i.e. each member of $|\phi| \cup |\psi|$ represents a distinct entity.

The second restriction is motivated by the principle that the number of distinct objects involved in a plan is a fundamental part of its specification. Identifying two different entities named in the specification of an atomic action produces a specification for an entirely different action, which should therefore have a distinct name. If the user includes two different entity symbols in a *StateSpec*, they should not be unified during the course of planning.

We combine these two restrictions into a single predicate: If A is a set of entity symbols (i.e. constants and variables), let $distinct(A)$ abbreviate the condition that the variables of A represent entities of the appropriate type, and that each element of A represents a different entity. If $A = \{a_i \mid i = 1, \ldots, n\}$ where the variables in A are $x_1, \ldots, x_m$ with types $\tau_1, \ldots, \tau_m$, then:

$$distinct(A) \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m} entity\_type(x_i, \tau_i) \wedge \bigwedge_{\substack{1 \le i,j \le n \\ i \neq j}} a_i \neq a_j$$

The definition of satisfaction for plan specifications must describe the change of state caused by executing the plan. This is expressed by the function $update\_state$, which maps a state σ and an interpretation $\hat{\phi}$ of a state specification ϕ to the new state that results from changing any attribute values in σ that disagree with $\hat{\phi}$. $update\_state(\sigma, \hat{\phi})$

is the state that maps each attribute name $a \in A_\tau$ to the function:

$$a^{update\_state(\sigma,\hat{\phi})}(\hat{e}) = \begin{cases} \hat{v} & \text{if } \exists\,(\hat{e}, a, \hat{v}) \in \hat{\phi} \\ a^\sigma(\hat{e}) & \text{otherwise} \end{cases}$$

This function is well-defined if ϕ is consistent and $\models_{\mathcal{M}} distinct(|\phi|)\,[\mathcal{V}]$ holds.

Finally, let the predicate $Sat_{\mathcal{M},\mathcal{V}}(\sigma, \phi)$ denote the satisfaction of state specification ϕ by state σ under interpretation $\mathcal{M}$ and valuation $\mathcal{V}$. It is defined by:

$$Sat_{\mathcal{M},\mathcal{V}}(\sigma, \phi) \equiv \forall\,(e, a, v) \in \phi \;\; a^\sigma(\hat{e}) = \hat{v}$$

Note that if ϕ is a consistent state specification and $\mathcal{M}$ and valuation $\mathcal{V}$ are such that $\models_{\mathcal{M}} distinct(|\phi|)\,[\mathcal{V}]$ holds, then ϕ is always satisfiable by some state, as the triples of ϕ generate a set of well-defined partial functions $a^\phi\colon {E_\tau}^{\mathcal{M}} \rightarrow |\mathcal{M}|$ for each attribute $a \in A_\tau$ (for some τ) appearing in ϕ. Any extension of these to total functions defines a state which satisfies ϕ (under $\mathcal{M}$ and $\mathcal{V}$).

We are now ready to define satisfaction for plan specifications:

Definition 5.2 *A plan specification* $\Theta \equiv \ulcorner \Gamma \triangleright \langle \phi, P, \psi \rangle \urcorner$ *is satisfied by interpretation* $\mathfrak{M} = \langle \mathcal{M}, \mathcal{F} \rangle$ *and valuation* $\mathcal{V}$*, written* $\models_{\mathfrak{M}} \Theta\,[\mathcal{V}]$*, if the following condition holds:*

$$\models_{\mathcal{M}} \Gamma\,[\mathcal{V}] \;\wedge\; \models_{\mathcal{M}} distinct(|\phi| \cup |\psi|)\,[\mathcal{V}]$$
$$\Rightarrow \quad \left[\forall \sigma \in \Sigma \;\; \forall \bar{p} \in I(P) \;\; Sat_{\mathcal{M},\mathcal{V}}(\sigma, \phi) \Rightarrow f_{\bar{p}}(\sigma) = update\_state(\sigma, \hat{\psi})\right]$$

■

The function *update_state* serves to build a frame axiom into the semantics of plan specifications, i.e. if an attribute is not assigned a new value in a plan specification then it continues to have its old value.

Note that this definition of satisfaction is not given solely in terms of the syntactic structure of the plan specification, but also depends on the names of the entity variable and constant symbols appearing in ϕ and ψ. This is due to the definition of the boolean expression abbreviated as $distinct(|\phi| \cup |\psi|)$. To avoid this problem, we consider a plan specification to have the form $\Gamma \triangleright \langle \phi, P, \psi \rangle_D$ where D is a multiset (or bag) specifying

which entity symbols must represent distinct elements for the plan specification to be meaningful. It is implicitly assumed that the multisets in the *PlanSpec* axioms describing the atomic actions contain one copy of each distinct entity symbol appearing in the formula, and that the conclusion of every inference rule is associated with the union of the multisets associated with the premise *PlanSpec* formulae (this is a *multiset* union, where a union $M \cup N$ is defined to have $\max(m, n)$ copies of each element e which appears m times in M and n times in N). The meta-logical expression $|\phi| \cup |\psi|$ in the definition above can then be replaced by D. Note that a plan specification for which D contains more than one copy of any element is always satisfied and therefore meaningless.

From this definition it is clear that for any precondition (represented by some triple (e, a, v) in ϕ) that still holds after the plan is executed, it is immaterial whether the triple appears in ψ or not. Therefore, the postcondition *StateSpec* in the body of a plan specification (i.e. ignoring the set of literals Γ) is not uniquely determined (up to renaming of variables) by the meaning of the *PlanSpec*. For example, it is always possible to re-express a consistent *PlanSpec* in a form $\langle \phi, P, \psi \rangle$ where $\phi \cap \psi = \varnothing$, but this has the same meaning as $\langle \phi, P, \xi \rangle$ for any ξ such that

$$\psi \subseteq \xi \subseteq \psi \cup \{(e, a, v) \in \phi \mid \nexists v'. (e, a, v') \in \psi\}$$

Therefore, to avoid requiring many different versions of the axioms specifying atomic actions, or inference rules[1] such as

$$\frac{\langle \phi, P, \psi \rangle}{\langle \phi, P, \psi \setminus (\phi \cap \psi) \rangle}$$

we follow the convention that the 'large' version of a plan specification is always used, i.e. for every triple $(e, a, v) \in \phi$, there must be a triple $(e, a, v') \in \psi$ for some v' — if the triple remains satisfied after the plan is executed, then we put $v' = v$. We will refer to this as the *maximal form* of a plan specification. This is the form that is required for planning, as we want to supply the initial goal *PlanSpec* in a form that contains *all* the triples describing the desired goal state, and not just those that are achieved by the last action in the plan. Of course, an implementation can automatically transform a

[1] Inference rules are written in the form $\frac{P_1 \ldots P_n}{C}$ where $P_1, \ldots, P_n$ are the premises and C is the conclusion.

PlanSpec supplied by a user into this form.

The inference rules must be defined so that they respect this convention: the conclusion must be in its maximal form whenever the *PlanSpecs* in the premises are.

If an interpretation $\mathfrak{M}$ satisfies a plan specification Θ for every valuation, then $\mathfrak{M}$ is said to be a *model* of Θ, written $\models_{\mathfrak{M}} \Theta$. A plan specification Θ is *valid* if it is satisfied by every interpretation and valuation, and we write $\models \Theta$.

A set Γ of *PlanSpec* formulae and formulae from the first order language $\mathcal{L}$ is said to *logically entail* a plan specification Θ if every interpretation that is a model of each formula in Γ is also a model for Θ. This is written as $\Gamma \models \Theta$.

5.2 Temporal Operators and Inference Rules

If a temporal operator ω is to be viable in a plan representation language, it should be possible to generate the plan specification formula for a plan $\omega(p_1, p_2, \ldots, p_n)$ from the specifications of the subplans $p_1, p_2, \ldots, p_n$. If this is the case, then the semantics for plan specifications given in the previous section can be used to derive an 'introduction rule' for ω. The *StateSpecs* appearing in the conclusion of this rule will be functions of those appearing in the premises. The planning system must therefore allow the user to define functions acting on *StateSpecs* such as union and difference ('\'), regarding them as sets of triples.

We will illustrate the use of inference rules in the case of linear planning, where only one temporal operator is needed: the sequential composition operator ';'. The following inference rule can be derived for this operator:

$$\frac{\Gamma \triangleright \langle \alpha, P, \beta \rangle \qquad \Delta \triangleright \langle \gamma, Q, \delta \rangle}{\Gamma, \Delta \triangleright \big\langle \alpha \cup (\gamma \setminus \beta),\ P;Q,\ update(\beta, \delta) \big\rangle} \quad \text{if } \beta \cup \gamma \text{ is a consistent } \textit{StateSpec}$$

where $update$ is a function that combines the triples of one state specification (ϕ) with the triples of another (ψ), updating any attribute values of ϕ that don't agree with those in ψ:

$$update(\phi, \psi) = \{(e, a, v) \mid (e, a, v) \in \phi \ \wedge \nexists v'. (e, a, v') \in \psi\} \cup \psi$$

This is similar to the function *update_state* used in the definition of *PlanSpec* satisfaction, except here the first argument is a state specification, not a state.

Note that this inference rule is a *schema* representing a whole class of valid inferences obtained by substituting object-level terms for the variables Γ, Δ, α, ..., δ, P and Q, such that the 'side condition' is met. In particular, the functions '$\cup$', '$\setminus$' and *update* are meta-level constructs that are used to describe the syntactic structure of the *StateSpecs* in the rule's conclusion as a function of those in the premises. In any instance of the rule, the conclusion *PlanSpec* will contain actual state specifications rather than functional expressions. However, as planning will be performed by using inference rules in a reverse direction to decompose (partially uninstantiated) plans into two or more subplans, the system will need to solve equations such as $\boldsymbol{\phi} = \boldsymbol{\alpha} \cup (\gamma \setminus \boldsymbol{\beta})$ and $\boldsymbol{\psi} = update(\boldsymbol{\beta}, \delta)$ where γ and δ are unknown, and the bold symbols represent non-variable terms. This process will be illustrated briefly in Section 5.3.3, and after discussing the relevant techniques in Chapter 6 and describing their application in our implemented planning system in Chapter 7, a detailed example is presented in Chapter 8.

A proof of the soundness of this inference rule is given in Appendix A.

The 'side condition' on the rule above is a syntactic restriction on the application of the rule: it is only valid if the effects (β) of the first plan (P) are consistent with the preconditions (γ) of Q. Note also that for the resulting plan specification to be meaningful, the preconditions for Q that are not achieved by P must be satisfied before the combined plans can be executed, and so must be consistent with the preconditions of P, i.e. $\alpha \cup (\gamma \setminus \beta)$ must be consistent. Unfortunately, these consistency requirements can only be checked for plan specification formulae containing fully evaluated state specifications, i.e. those containing no variables representing unknown *StateSpecs* or function symbols such as '$\cup$', and there is no obvious way to prevent variables becoming instantiated in a way that would violate these constraints. This is unlike the common forms of syntactic restriction that arise in logical systems, such as in the $\forall$-introduction rule discussed by Paulson (1986):

$$\frac{\Gamma \vdash A[x]}{\Gamma \vdash \forall y.A[y]} \qquad x \text{ not free in } \Gamma$$

This restriction on the possible bindings for x can be enforced by representing it as a Skolem function (see Section 5.3.5).

In the case of the consistency requirement above, the semantics of plan specifications make it possible to use a stronger *semantic* consistency constraint to ensure that the application of an inference rule will not become invalidated at a later stage of planning due to the instantiation of state specification variables. Consider a state specification expression S that is constrained to be consistent for some inference rule to be applicable. Then in the completed proof, when S has been instantiated (by some substitution σ) and evaluated to give a fully evaluated *StateSpec* term $\rho = \{(e_1, a_1, v_1), \ldots, (e_n, a_n, v_n)\}$, one of the (syntactic) conditions $e_i \not\equiv e_j$ or $a_i \not\equiv a_j$ must hold for each i and j $(i \neq j)$. This can be guaranteed by applying the inference rule under the assumption $consistent(S)$, where the predicate $consistent$ is defined by:

$$consistent(\phi) \stackrel{\text{def}}{\equiv} \bigwedge_{\substack{1 \leq i,j \leq n \\ i \neq j}} (e_i \neq e_j \vee a_i \neq a_j)$$

with equality on attribute names interpreted as syntactic identity (the attribute names are uninterpreted in the semantics — they are simply syntactic place-holders). In an implementation, this function can be defined recursively in terms of the structure of state specifications, and the assumption $consistent(S)$ above will eventually reduce (after instantiation) to a conjunction of the 'disequations' $e_i \neq e_j$ for each i, j such that $a_i = a_j$. These assumptions may not be able to be discharged in the final proof state as one or both of e_i and e_j may remain uninstantiated in the final *StateSpec* term ρ. This could result in the proof being invalid for any interpretation $\mathcal{M}$ and valuation $\mathcal{V}$ which assign the same value to e_i and e_j; however, the semantics of plan specifications guarantee that this is not the case: if e_i and e_j are syntactically different, then they will both appear in the *StateSpecs* of the proven goal formula, and the premise $distinct(|\phi| \cup |\psi|)$ appearing in the definition of *PlanSpec* satisfaction ensures that the goal formula is satisfied (trivially) by $\mathcal{M}$ and $\mathcal{V}$.

Similarly, to avoid generating valid but meaningless *PlanSpecs*, the whole proof can be conducted under the initial assumption that $distinct(|\phi| \cup |\psi|)$ holds (recall that this is a meta-level expression denoting a conjunction of disequations), and the semantics

guarantee that the proven *PlanSpec* will hold regardless of whether the distinct entity symbols are instantiated to become distinct constants or not.

In the current system, the disequality constraints are implemented using the SICStus Prolog predicate dif/2. Calls to this predicate are delayed until the goal is sufficiently instantiated and then succeed or fail immediately (this treatment of negation relies on the *closed-world* assumption which is discussed in Section 5.3.2). Any dif/2 constraints that are still pending when planning is completed can be safely ignored for the reasons outlined above. The state specification data structure is implemented using conditional rewrite rules, some of which have conditions requiring pairs of entities to be unequal. Therefore, the disequality assumptions can be used constructively to advance the inference process.

As we are only interested in deriving plan specifications that are (syntactically) consistent, a semantic consistency assumption can be added to the proof state for each new state specification term that is generated during the planning process. This is equivalent to introducing a new state specification false which can never be satisfied, and defining '$\cup$' and '$\setminus$' to be *strict* functions (producing the value false if either argument is false). Planning would then involve proving a theorem of the form $\langle \phi, P, \psi \rangle$ under the constraints $\phi \neq \mathsf{false}$ and $\psi \neq \mathsf{false}$. This gives the behaviour wanted in an implementation, where the generation of an inconsistent *StateSpec* should cause the planner to fail and backtrack.

An alternative approach is used in the system described in Chapter 7: state specifications are defined using *order-sorted* equational logic, which provides an elegant way of defining functions that are restricted to some equationally-defined subset of a data type. This technique, and others that are used to implement inference in this logic, will be described in the next chapter.

5.3 Planning in a Logic of Plan Specifications

In the framework presented above the specification of the plan to be found is given as a formula which must be shown to be true using a number of inference rules and assuming the specifications of the atomic actions as axioms, i.e. the goal *PlanSpec* is

a theorem to be proved. However, planning is a difficult problem, especially when arbitrary temporal operators may appear in plans, and we cannot expect to have a general-purpose theorem prover to automate this process. As new temporal operators are defined, the user must be able to design and test appropriate planning strategies to determine when these operators should be used in a plan. In this section we discuss a technique used in a number of interactive theorem provers to provide this level of flexibility.

5.3.1 Tactical Theorem Proving

The Edinburgh LCF system [Gordon, Milner & Wadsworth 79] introduced a new approach to interactive theorem proving by providing a *meta-language*, interfaced with the formal logic, in which the user can write programs to perform part of the proof. This technique is now well-established in theorem provers designed to assist the development of proofs in expressively powerful logics, such as Cambridge LCF [Paulson 87a], Nuprl [Constable et al. 86] and the Edinburgh reconstruction of Nuprl [Horn 88], now known as the Oyster theorem prover. These systems are built around a natural deduction style proof system [Tennant 78, Barwise 77], where the emphasis is on using 'natural' inference rules and simple axioms, rather than a single inference rule (such as resolution or modus ponens) with many axioms. Inference rules are treated as functions producing theorems from other theorems. The goal is proved by building a proof-tree where the leaves are axioms, the internal nodes are theorems proved by applying inference rules to their children, and the root is the desired goal. Most proofs are conducted backwards, using goal-decomposing *tactics*. These are functions designed to assist the proof of a goal theorem by decomposing it into a number of subgoals.

Each tactic also returns a *validation function* describing how a proof of the goal can be generated from proofs of the subgoals. In general, a tactic is designed to apply to goals of a certain form, and will fail if incorrectly used. Tactics can be combined to form more powerful tactics using higher-order functions called *tacticals*. This allows powerful proof strategies to be expressed using control constructs such as sequencing, recursion, and the conditional application of tactics (based on the success or failure of another tactic) [Schmidt 84]. Tactics can also be used to rewrite goals to a simpler form

[Paulson 83]. In LCF style theorem provers, users can design their own tactics and tacticals using the system's meta-language (ML for LCF and Nuprl, Prolog for Oyster) and use these to automate part (or all) of the proof process.

Application of a tactic to a goal is usually done by matching the parameter of the tactic to the goal. However, for the planning logic presented above, the plan is generated by instantiating the plan term in the goal plan specification. This requires unification to be used when extending the partial proof tree by the application of inference rules. Sokolowski (1983) extended Edinburgh LCF by allowing *pattern variables* in goals, which could be instantiated by tactics. This technique has advantages when using rules involving quantifiers, such as '$\exists$-Introduction', presented here in its simplest form:

$$\frac{A[t]}{\exists x\ A[x]}$$

where $A[x]$ is a formula with free occurrences of x, and $A[t]$ is the result after substituting those occurrences with t.

In backwards proof without unification, the corresponding tactic must be provided with an extra argument specifying which value to use for t. However, an appropriate value may not become obvious until later in the proof. Unification allows this decision to be delayed until a correct value can be determined, and the chosen value will then be propagated throughout the proof tree. The same advantages apply to pattern variables appearing in the original goal, as well as for those introduced by inference rules.

Paulson (1986) claims that adding unification to tactics places inference rules in the dominant position, rather than the theorems they manipulate. His theorem prover, Isabelle, treats inference rules as propositions about the premises and conclusion, forming proofs by composing inference rules, thus providing a single model for forwards and backwards proof. Paulson's tactics are functions on inference rules, which are regarded as partial proof trees. The final proof tree is a derived inference rule where all the leaves are axioms. In Isabelle, an inference rule can be seen as a Horn clause in a higher-order logic, and inference rules can therefore be composed by resolution. This notion is formalised in [Paulson 87b]. There is one basic difference from other systems using resolution: in resolution-based theorem provers such as

Prolog, the theorem to be proved is a goal clause, i.e. a clause of the form '$G \Rightarrow$', whereas in Isabelle, the initial proof state is the trivial inference rule $\frac{G}{G}$ (represented in the higher-order logic as $G \Rightarrow G$), where G is the goal theorem. The initial goal may contain variables to be instantiated during the proof. This is needed for existence proofs, such as Paulson's example of proving an algorithm runs in linear time by showing it runs in Kn time units for some K, where n is the size of the input. We cannot expect to know K at the start of the proof. K can be expressed as a *scheme variable* which will be instantiated at an appropriate stage of the proof. Universally quantified variables can be represented in the goal by Skolem constants (although a later version of Isabelle uses a different method [Paulson 87b]).

In the next section we show how these ideas can be applied in the planning domain.

5.3.2 Planning with Inference Rules

Section 5.1.1 introduced plan specification formulae as a means of describing the pre- and postconditions of atomic actions and plans. The same type of formula can be used to represent the specification of the desired plan: given state specifications ϕ and ψ describing the initial and goal states respectively, the aim of the planner is to find a plan P such that the formula $\langle \phi, P, \psi \rangle$ holds. In this framework, planning proceeds by decomposing *PlanSpec* goals into subgoals using the temporal operator 'introduction' inference rules in the reverse direction. An axiom A is treated as an inference rule $\frac{}{A}$ with an empty premise, so any subgoal that unifies with an axiom can be resolved away. However, using the user-defined plan specification axioms in this way requires a generalisation of the notion of an inference rule. As Loeckx and Sieber (1987) discuss, a logical calculus of axioms and inference rules is simply a system for deriving syntactic objects from other syntactic objects. Of course, these rules are usually intended to be used for the derivation of formulae having some common semantic property (usually that of logical validity), but the particular property desired depends on the application. In our case, we wish to prove plan specification formulae Θ for which the following property holds:

$$Ax \cup \mathbf{C}^* \models \Theta$$

where Ax is the set of *PlanSpec* axioms describing the atomic actions and C* is the *logical completion* of the set of Horn clauses C, i.e. C augmented with the negation of every positive ground literal that does not logically follow from C. This states that Θ is a logical consequence of the axioms defining the atomic actions together with the Prolog program provided by the user. Clearly, all instances of the axioms in Ax have this property, and for the inference procedure to be sound, the conclusion of each inference rule must have this property whenever the premises do (although, in general the rules will satisfy a stronger condition: that of preserving satisfaction for every interpretation and valuation).

Reasoning with C* corresponds to making a *closed-world* assumption, where the state of the world is assumed to be described completely by the set of clauses C and any proposition that does not follow from these clauses must therefore be false. Equivalently, a proposition is considered to be true precisely when it is satisfied by the least Herbrand model (or initial model—see Section 6.2.1) of C. This is the standard model-theoretic semantics for Prolog [van Emden & Kowalski 76] and ensures the soundness of interpreting logical negation as 'negation as failure' [Clark 78]. In particular, under this interpretation the entities represented by two distinct entity constants are provably not equal.

In this framework, planning is a search through the space of proof states generated by four types of choice. For each inference step, the planner must choose: (1) a subgoal to decompose; (2) an inference rule to apply (in reverse); and (3) which of the possible unifiers of the subgoal and the conclusion of the rule to use. If the selected subgoal is a Prolog goal then the system must select: (4) an answer substitution for the Prolog query. The search is controlled by using tactics to guide the first three types of choice and determine when and how to use backtracking. The satisfaction of Prolog goals is controlled using the normal computation and backtracking rules (i.e. by calling them directly in a Prolog implementation). The planning process succeeds if there are no subgoals remaining. At this stage, the initial plan term P has become instantiated to give the desired plan.

The planner starts with the trivial proof tree

$$\frac{\langle \phi, P, \psi \rangle}{\langle \phi, P, \psi \rangle}$$

and at any stage it has a current proof tree of the form

$$\frac{\begin{array}{ccccc} S_1 & \cdots & S_i & \cdots & S_n \\ & \ddots & \vdots & \iddots & \end{array}}{G}$$

where the root G is the original goal formula and the leaves $S_1, \ldots, S_n$ are the current subgoals.

A planning step involves finding an inference rule

$$\frac{A_1 \ldots A_m}{B}$$

such that the conclusion B unifies with one of the subgoals S_i with a unifier θ. Composing this rule with the current proof tree produces the new tree:

$$\frac{\begin{array}{ccccccccc} & & & & \theta A_1 \cdots \theta A_m & & & & \\ \theta S_1 & \cdots & \theta S_{i-1} & & \theta S_i & & \theta S_{i+1} & \cdots & \theta S_n \\ & \ddots & & \ddots & \vdots & \iddots & & \iddots & \end{array}}{\theta G}$$

If the plan term of the *PlanSpec* S_i is a variable P, and the inference rule for ω-introduction is used (for some temporal operator ω) then S_i is bound to $\theta B = \omega(\theta P_1, \ldots, \theta P_n)$. As S_i is a subterm of the plan term of G, this results in the representation of the desired plan becoming further instantiated.

The internal state of a proof tree does not affect the proof process. Therefore, we can represent the application of an inference rule as a transformation on a proof *state* which records only the desired conclusion and current subgoals:

$$\frac{S_1 \ldots S_i \ldots S_n}{G} \Longrightarrow \frac{\theta S_1 \ldots \theta S_{i-1}\ \theta A_1 \ldots \theta A_m\ \theta S_{i+1} \ldots \theta S_n}{\theta G}$$

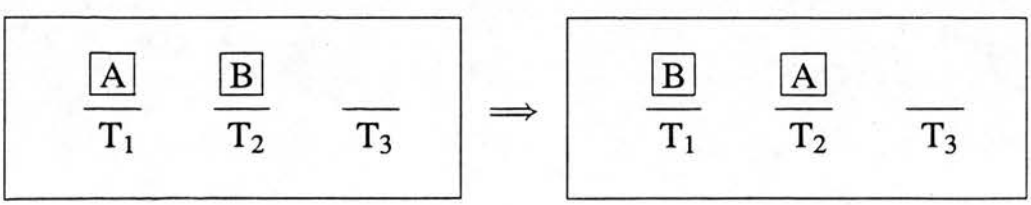
A proof state can be seen as a derived inference rule, and when no subgoals remain,

the planner will have succeeded in deriving a rule $\frac{}{G'}$ where $G' \equiv \ulcorner\langle\phi', P', \psi'\rangle\urcorner$ is an instance of the original goal formula G. P' is the resulting plan.

5.3.3 The Planning Strategy

In interactive theorem provers such as LCF, the strategy used to prove a goal formula depends on the structure of that goal. The tactics corresponding to each inference rule will only succeed in decomposing goals which have the appropriate form, and tacticals such as 'ORELSE' [Schmidt 84] allow the success or failure of a tactic to guide the proof process. Subgoal filtering constructs can be used to focus the search by placing restrictions on the form of subgoals that may be considered for reduction. In the planning domain, however, the form of the desired plan is usually unknown and therefore the structure of a plan specification provides little information about how it might be proved. Instead, the search for a proof must be guided by information about the initial and final states for each subgoal (and therefore the tactics used might more appropriately be called strategics!).

In the current plan state, each subgoal gives a partial specification of a plan that will help to solve the original planning problem. The instantiated parts of the pre- and postcondition *StateSpecs* can be used to select an action that will partially satisfy the specification for this subplan. This action is introduced into the plan by decomposing the goal into a specification for the chosen action and specifications for the other parts of the subplan. The way the action is selected dictates the type of planning strategy that is used. For example, consider the sequential planning problem of swapping the positions of two blocks in the simple blocks world discussed in Section 4.2.1, where there is one generic action — the *move* operator.



Initial State Final State

This problem is described by the initial goal:

$$\left\{\begin{array}{l}\text{loc}(A) = T_1\\ \text{loc}(B) = T_2\\ \text{on}(T_1) = A\\ \text{on}(T_2) = B\\ \text{on}(T_3) = \text{nil}\end{array}\right\} \quad P \quad \left\{\begin{array}{l}\text{loc}(A) = T_2\\ \text{loc}(B) = T_1\\ \text{on}(T_1) = B\\ \text{on}(T_2) = A\\ \text{on}(T_3) = \text{nil}\end{array}\right\}$$

To illustrate how planning works in this logical framework we consider a simple backwards planning strategy (this corresponds to the traditional planning technique of *goal regression*; see [Genesereth & Nilsson 87] or [Georgeff 87]). Using this strategy, planning proceeds by selecting an action A that achieves one or more of the goals (represented by triples) of the goal state specification, determining what additional preconditions must hold to ensure that the remaining goals are also satisfied after the execution of this action, and combining these with the action's preconditions to form the goal state specification for a new *PlanSpec*. Solving this recursively generates a subplan P such that the sequence P ; A solves the original problem. The actions in the plan are therefore generated in the reverse order.

In this example, there are only two possible final actions: $\text{move}(A, T_3, T_2)$ or $\text{move}(B, T_3, T_1)$. Choosing the first of these, we instantiate the ';'-introduction rule to get:

$$\frac{\langle \alpha, P, \beta \rangle \qquad \langle \gamma,\ \text{move}(A, T_3, T_2),\ \delta \rangle}{\Big\langle \alpha \cup (\gamma \setminus \beta),\ P\,;\text{move}(A, T_3, T_2),\ \mathit{update}(\beta, \delta) \Big\rangle}$$

where $\gamma \equiv \{\text{loc}(A) = T_3,\ \text{on}(T_3) = A,\ \text{on}(T_2) = \text{nil}\}$
and $\delta \equiv \{\text{loc}(A) = T_2,\ \text{on}(T_3) = \text{nil},\ \text{on}(T_2) = A\}$.

The conclusion of this rule is unified with the initial goal, which involves solving the

the two equations:

$$\alpha \cup (\gamma \setminus \beta) \;=\; \left\{ \begin{array}{l} \mathrm{loc}(A) = T_1 \\ \mathrm{loc}(B) = T_2 \\ \mathrm{on}(T_1) = A \\ \mathrm{on}(T_2) = B \\ \mathrm{on}(T_3) = \mathrm{nil} \end{array} \right\}$$

and

$$\mathit{update}(\beta, \delta) \;=\; \left\{ \begin{array}{l} \mathrm{loc}(A) = T_2 \\ \mathrm{loc}(B) = T_1 \\ \mathrm{on}(T_1) = B \\ \mathrm{on}(T_2) = A \\ \mathrm{on}(T_3) = \mathrm{nil} \end{array} \right\}$$

Equations such as this cannot be solved completely; instead they are solved as far as possible and any remaining equality constraints are kept as part of the proof state. In this example, we can deduce from the first equation that $\beta = \{\mathrm{loc}(A) = T_3,\ \mathrm{on}(T_3) = A,\ \mathrm{on}(T_2) = \mathrm{nil}\} \cup X$ (where X is a new *StateSpec* variable representing the remaining unknown part of β) as the right hand side is not consistent with the triples of γ and therefore it follows that these triples must appear in β. Also, with this value of β, the expression $\gamma \setminus \beta$ reduces to the empty set and so the left hand side of the first equation reduces to the variable α which can now be instantiated to solve this equation. The second equation implies that β (and therefore X) must contain the triples $\mathrm{loc}(B) = T_1$ and $\mathrm{on}(T_1) = B$. These bindings for α and β suggest choosing $\mathrm{move}(B, T_2, T_1)$ as the final action for the subplan P, and this can be used to guide the decomposition of the subgoal $\langle \alpha, P, \beta \rangle$. By a similar process, this leads to a new plan specification goal which can be satisfied by the *PlanSpec* axiom corresponding to the action $\mathrm{move}(A, T_1, T_3)$. The problem is now solved giving the proof tree shown in Figure 5.1. ∎

The system described in Chapter 7 implements the functions on state specifications using conditional rewrite rules in an equational logic. Equations are solved by reducing the two sides to their normal forms, applying user-defined transformations that map

$$\dfrac{\dfrac{\left\{\begin{array}{l}\mathrm{loc}(A)=T_1\\ \mathrm{on}(T_1)=A\\ \mathrm{on}(T_3)=\mathrm{nil}\end{array}\right\}\ \mathrm{move}(A,T_1,T_3)\ \left\{\begin{array}{l}\mathrm{loc}(A)=T_3\\ \mathrm{on}(T_1)=\mathrm{nil}\\ \mathrm{on}(T_3)=A\end{array}\right\}\qquad \dfrac{\left\{\begin{array}{l}\mathrm{loc}(B)=T_2\\ \mathrm{on}(T_2)=B\\ \mathrm{on}(T_1)=\mathrm{nil}\end{array}\right\}\ \mathrm{move}(B,T_2,T_1)\ \left\{\begin{array}{l}\mathrm{loc}(B)=T_1\\ \mathrm{on}(T_2)=\mathrm{nil}\\ \mathrm{on}(T_1)=B\end{array}\right\}}{\left\{\begin{array}{l}\mathrm{loc}(A)=T_1\\ \mathrm{loc}(B)=T_2\\ \mathrm{on}(T_1)=A\\ \mathrm{on}(T_2)=B\\ \mathrm{on}(T_3)=\mathrm{nil}\end{array}\right\}\ \mathrm{move}(A,T_1,T_3);\mathrm{move}(B,T_2,T_1)\ \left\{\begin{array}{l}\mathrm{loc}(A)=T_3\\ \mathrm{loc}(B)=T_1\\ \mathrm{on}(T_1)=B\\ \mathrm{on}(T_2)=\mathrm{nil}\\ \mathrm{on}(T_3)=A\end{array}\right\}}\qquad \dfrac{\left\{\begin{array}{l}\mathrm{loc}(A)=T_3\\ \mathrm{on}(T_2)=\mathrm{nil}\\ \mathrm{on}(T_3)=A\end{array}\right\}\ \mathrm{move}(A,T_3,T_2)\ \left\{\begin{array}{l}\mathrm{loc}(A)=T_2\\ \mathrm{on}(T_2)=A\\ \mathrm{on}(T_3)=\mathrm{nil}\end{array}\right\}}{\left\{\begin{array}{l}\mathrm{loc}(A)=T_1\\ \mathrm{loc}(B)=T_2\\ \mathrm{on}(T_1)=A\\ \mathrm{on}(T_2)=B\\ \mathrm{on}(T_3)=\mathrm{nil}\end{array}\right\}\ \mathrm{move}(A,T_1,T_3);\mathrm{move}(B,T_2,T_1);\mathrm{move}(A,T_3,T_2)\ \left\{\begin{array}{l}\mathrm{loc}(A)=T_2\\ \mathrm{loc}(B)=T_1\\ \mathrm{on}(T_1)=B\\ \mathrm{on}(T_2)=A\\ \mathrm{on}(T_3)=\mathrm{nil}\end{array}\right\}}$$

Figure 5.1: The final proof tree for a proof using a backwards planning strategy.

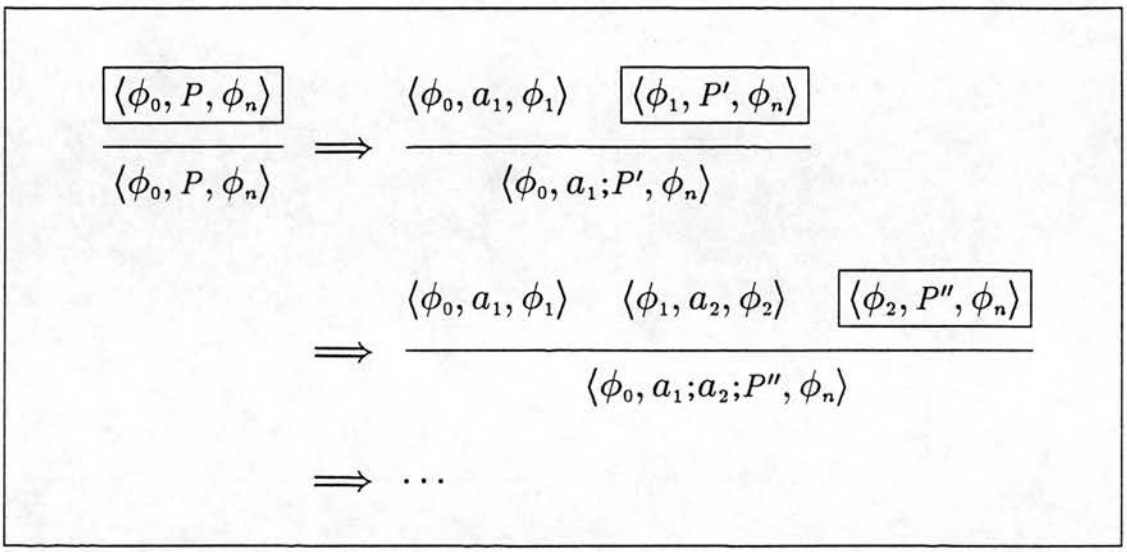
equations to an equivalent set of simpler equations (these are defined as rewrite rules that convert an equation to a conjunction of equations, but new variables may appear on the right hand side to allow the communication of partial answer substitutions between the different conjuncts), or by partially unifying the two sides to produce a substitution and a set of equations representing the unsolved part of the unification problem. In Chapter 8, these techniques are illustrated by working through the solution of the block-swapping problem in detail, using two different strategies: planning forwards from the initial state, and the backwards planning strategy described above (although the discussion there does not correspond exactly to the sketch given above as the initial plan specification is given in a more general form — which turns out to be equivalent in this case). Figure 5.2 illustrates the sequence of proof states obtained using these two strategies.

5.3.4 Evaluating Prolog Goals

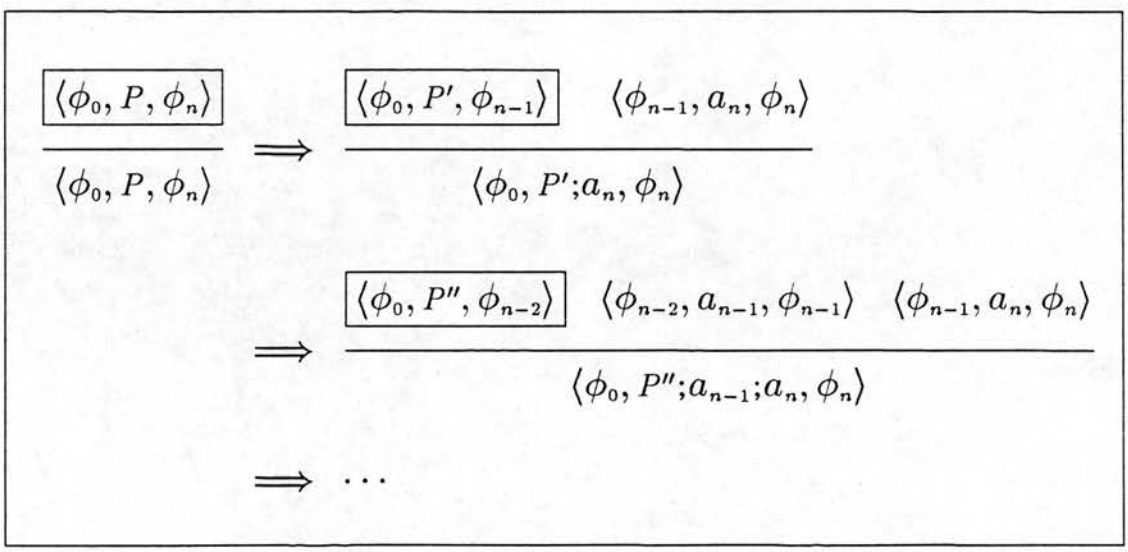
The discussion above treats the plan specification goals as atomic formulae, with no consideration of their internal structure. However, recall that the *PlanSpec* axioms describing the atomic actions may include as hypotheses a set of literals in the object level language $\mathcal{L}$, i.e. Prolog goals. These are intended to be evaluated during planning to test complex preconditions and to construct terms that appear in the postconditions. These goals do not play any other role during planning, and in particular, we do *not* use any inference rules involving general predicate logic deduction such as:

$$\frac{\Gamma, B \triangleright \langle \phi, P, \psi \rangle \qquad A \rightarrow B}{\Gamma, A \triangleright \langle \phi, P, \psi \rangle}$$

When a current goal in the proof tree is unified with an atomic action axiom, the associated literals should be added as new subgoals, which will be evaluated during the course of planning. The initial goal may also include a set of Prolog goals Δ to express any extra conditions on the plan to be generated, and to generate terms appearing in the pre- and postconditions of the initial plan specification. These are treated as if the goal to be proved was of the form $\Delta \wedge \langle \phi, P, \psi \rangle$, although the conjunction symbol '$\wedge$' is not part of our language. In an implementation this treatment of literals can be achieved



(a) A forwards planning strategy.



(b) A backwards planning strategy.

Figure 5.2: Forwards and backwards planning strategies using the ';'-introduction rule. The boxed subgoal at each step is the next to be decomposed; the remaining subgoals are instances of the axioms describing the atomic actions, but are left in the proof state for illustrative purposes. For simplicity it is assumed that the state specifications ϕ_i do not become instantiated during unification, and that no axioms involve Prolog goals.

simply by maintaining a separate set of current Prolog goals, to which new literals are added when atomic action axioms are used, and whose members may be evaluated at any stage of planning. Formally, this corresponds to using the initial proof tree:

$$\frac{\Delta, \Gamma \qquad \Gamma \triangleright \langle \phi, P, \psi \rangle}{\Delta \wedge \langle \phi, P, \psi \rangle}$$

or equivalently, applying this rule as the first step of the planning process. The subgoal $\Gamma \triangleright \langle \phi, P, \psi \rangle$ will be decomposed during the course of planning as described above. Γ will initially be a variable, but will become successively more instantiated as planning proceeds. The other subgoal, "Δ, Γ", is treated as a (partially instantiated) conjunction $g_1 \wedge \cdots \wedge g_n \wedge \Gamma$ where $\Delta = \{g_1, \ldots, g_n\}$.

At any stage a literal can be passed to the Prolog interpreter to be executed. If the query succeeds, the literal is removed from the proof state (i.e. it has been resolved away) and the resulting substitution is applied to the other subgoals and the conclusion. This is expressed by the following proof state transformation:

$$\frac{\Gamma, A \quad S_1 \cdots S_n}{G} \implies \frac{\theta\Gamma \quad \theta S_1 \cdots \theta S_n}{\theta G}$$

where θ is an answer substitution for the query '?- A' with respect to the Prolog program C.

This transformation corresponds to applying the inference rule $\frac{}{\theta A}$ which is sound as the success of the query shows that $\mathbf{C}^* \models \theta A$ (and therefore the weaker condition $Ax \cup \mathbf{C}^* \models \theta A$) must hold.

5.3.5 Variables, Instantiation and Quantification

A goal appearing in the current proof state may contain variables denoting plan terms, *StateSpec* and *PlanSpec* terms and sets of literals. These are not part of the object-level logic, but belong to the meta-language in which we derive inference rules. This raises questions about the semantics of our proof system: how can we understand the use of these meta-level variables? How do they differ from the object-level variables, and when can they be instantiated? In this section we briefly discuss the developments in

interactive theorem proving that address these issues.

The inference rules of a logical system are *schemas* representing a whole class of valid inferences, i.e. all instances that can be obtained by instantiating the variables of the rule with actual terms and formulae. In the theorem prover LCF, variables may only range over object-level terms. Inference rules are implemented as functions which construct new theorems from existing theorems.

Isabelle-86 introduced *scheme* variables which can appear in the initial goal to denote formulae, and allow generic theorems and inference rules to be derived. Scheme variables are also used to fill-in for object-level terms which are unknown at the start of the proof. When the current proof state is resolved with an inference rule, these variables may become instantiated, allowing an appropriate value for a term to be discovered during the proof and then propagated throughout the proof tree.

In Paulson's formal reconstruction of the ideas behind Isabelle [Paulson 87b], the inference rules of the object logic and the current proof state are expressed as implications in a higher-order typed logic (HOL). For instance, a conjunction introduction rule is represented as

$$\mathrm{true}(A) \Longrightarrow \mathrm{true}(B) \Longrightarrow \mathrm{true}(A \wedge B)$$

where $\mathrm{true}(F)$ denotes the proposition that the object logic formula F is true. Proof proceeds by resolving the proof state with an inference rule, producing a new proof state. All variables belong to the higher-order logic, and their types determine whether they denote terms, object-level formulae or propositions about formulae. Any variable can be instantiated during the proof, provided it does not occur free in any undischarged assumptions. This provides a semantics for the proof state transformations described in the previous section, although unlike the later versions of Isabelle we do not reason explicitly in higher-order logic.

It may seem that allowing any variables in the goal to become instantiated during planning will result in plans that are too specialised. However, resolving a proof state with an inference rule will generally only partially instantiate the variables of the proof state, binding them to other variables or terms involving the variables of the new subgoals. Only the plan terms will become totally instantiated during planning. The axioms, and therefore the final plan specification, will be generic, with variable entity

and object terms appearing in their pre- and postcondition *StateSpecs*. When there are atomic actions whose specifications involve particular objects, it may be necessary to avoid undesired instantiation by including universally quantified variables in the initial goal. We can achieve the same result in a language without quantifiers by using Skolem functions.

Consider a goal G containing one or more occurrences of the free variable x. Let $G[t]$ denote the term obtained by substituting t for all occurrences of x in G. We will sometimes write $G[x]$ for G itself to emphasize the dependence on x. To prove $\forall x\, G[x]$ we must show that $G[y]$ is true for an arbitrary term y. The actual term we use is not important, as long as the proof of $G[y]$ does not depend on any particular property of the chosen y, and y cannot appear elsewhere in G. We avoid these potential problems by introducing a new function symbol y and attempting to prove the goal $G[\mathrm{y}(v_1, \ldots, v_n)]$ where $v_1, \ldots, v_n$ are the free variables of G. No free variable v_i in this goal can become 'accidentally' instantiated to a term containing an instance of $\mathrm{y}(v_1, \ldots, v_n)$ as the resulting term would be circular. This, of course, relies on the 'occurs check' being present in the unification algorithm.

In Isabelle-86, a Skolem function is treated as a constant $\mathrm{y}_{v_1,\ldots,v_n}$ with the free variables v_i considered to be part of the constant's name. This is implemented by generating a unique name for the constant and associating it with a directed acyclic graph that records its dependencies on the variables v_i. As these variables become instantiated, the graph grows, and any variable bindings that would introduce a cycle are disallowed. This technique is due to Wallen (1985).

The addition of the explicit meta-logic HOL to Isabelle allowed a more elegant treatment of the restrictions on 'arbitrary' values in proofs. By expressing the universal quantification in the meta-level, and using a technique called $\wedge$-lifting ($\wedge$ represents universal quantification in HOL), the use of Skolem constants can be avoided. This technique, however, relies on the use of higher-order unification that is required in Paulson's framework.

Let us now consider a typical planning goal $G[\bar{x}, \bar{y}]$ where we wish to find particular values $\bar{t}$ for the vector of variables $\bar{x}$ such that for all values of the variables $\bar{y}$, the plan specification $G[\bar{t}, \bar{y}]$ holds. We can represent this by the quantified goal $\exists \bar{x}\, \forall \bar{y}\, G[\bar{x}, \bar{y}]$.

Without quantifiers, we must leave the elements of $\bar{x}$ as free variables which may become instantiated during planning. It doesn't matter if any x_i is not instantiated — we will have proved a result stronger than the requested goal. The universally quantified variables $\bar{y}$ must be replaced by Skolem functions $\mathrm{y}(\bar{x})$, parameterised by the variables $\bar{x}$ to prevent any y_i appearing as a subterm of some x_j. This gives us the initial proof state

$$\frac{G[\bar{x}, \bar{\mathrm{y}}(\bar{x})]}{G[\bar{x}, \bar{\mathrm{y}}(\bar{x})]}$$

An alternative way to view this use of Skolemization is to consider the initial proof state $\frac{G}{G}$. This represents the meta-level implication $\mathrm{true}(G) \Longrightarrow \mathrm{true}(G)$ or equivalently $\neg\mathrm{true}(G) \vee \mathrm{true}(G)$. This will follow if we can prove $\mathrm{true}(\neg G) \vee \mathrm{true}(G)$. The proof of G proceeds by resolving away $\mathrm{true}(\neg G)$, leaving the final proof state $\frac{}{G}$. Now, $\neg G \equiv \forall \bar{x}\, \exists \bar{y}\, \neg G(\bar{x}, \bar{y})$ and by replacing the existentially quantified variables with Skolem functions, in the usual technique for refutation theorem provers, we get the negative literal (i.e. goal clause) $\neg G[\bar{x}, \bar{\mathrm{y}}(\bar{x})]$. This gives us the same initial goal as the argument in the previous paragraph.

Chapter 6

Computing with State Specifications: Implementation Techniques

6.1 Introduction

The main issue in implementing the planning framework presented in Chapter 5 is providing a mechanism for computing with state specifications. These are effectively sets of ⟨entity, attribute, value⟩ triples, with some additional constraints (e.g. all elements must be distinct[1]). The terms representing state specifications that are manipulated by the planner may be only partly instantiated and could contain function symbols such as union and difference that are introduced by the inference rules associated with each temporal operator. To allow new temporal operators to be used it should be easy to define functions that act on *StateSpecs*, and to have terms involving these function symbols that occur in the proof state to be reduced whenever they are sufficiently instantiated. Also, *StateSpecs* must be unified during the course of planning. The unification algorithm should ignore the ordering of triples in a *StateSpec*, and so there may be multiple unifiers. However, no general unification algorithm can be provided to deal with arbitrary user-defined functions on *StateSpecs*, so there must be support for delaying unification until these functions have been evaluated (although a mechanism for early detection of failure would be very useful).

[1]This is not the case with sets, e.g. the set $\{X, Y, 2, 3\}$ could be instantiated to $\{1, 2, 3\}$ by the substitution $\{1/X, 1/Y\}$.

These criteria are satisfied by defining *StateSpecs* as abstract data structures using *order-sorted* equational logic, and by using a *multi-valued mode system* that enables preconditions to be associated with the unification algorithm for each data type, delaying unification of two terms until they are sufficiently instantiated.

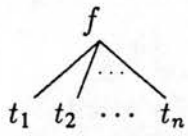
6.2 Abstract Data Types and Order-Sorted Algebra

To specify a data type, it is necessary to define the set of possible objects of that type and the functions that act on them. The principle of *data abstraction* states that this should be done independently of any concrete representation, and a theoretical basis for this idea is provided by the mathematical notion of a *many-sorted algebra*. Given a set of *sorts* S (corresponding to the data types of interest), a (many-sorted) algebra $\mathcal{A}$ consists of a family $\{\mathcal{A}_s \mid s \in S\}$ of sets of objects for each sort, and a family of functions (or *operators*) $\mathcal{F}$ mapping tuples of objects to objects. $\mathcal{A}_s$ is called the *carrier* of sort s in the algebra.

An algebra is usually defined relative to a *signature* which specifies the names and the argument and result types of the functions of $\mathcal{A}$. Mathematically, this is a family $\Sigma = \{\Sigma_{\bar{w},s} \mid \bar{w} \in S^*, s \in S\}$ of sets (of function symbols) for each sequence of argument sorts $\bar{w}$ and sort s. A function symbol $f \in \Sigma_{\bar{w},s}$ is said to have *arity* $\bar{w}$ and *sort* (or sometimes *coarity*) s. The pair $\langle \bar{w}, s \rangle$ is said to be a *rank* of f. Note that a function symbol may have more than one associated rank.

A Σ-algebra $\mathcal{A}$ is an algebra such that there is a one to one mapping between the function symbols of Σ and the functions of $\mathcal{A}$ that respects the arities and sorts of the function symbols. The functions of $\mathcal{A}$ are usually considered to be indexed by the corresponding function symbols, i.e. $\mathcal{F} = \{\mathcal{F}_F \mid F \in \bigcup_{\bar{w},s} \Sigma_{\bar{w},s}\}$ where $\mathcal{F}_F\colon A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ if $F \in \Sigma_{(s_1 \ldots s_n),s}$.

Given a signature Σ and a sort-indexed set of variable symbols X, we can use the function and variable symbols themselves to form a Σ-algebra of terms (regarded as syntax trees) called the *term algebra* and denoted $T_\Sigma(X)$. The carriers of each sort are the constants and variables of that sort, and the function corresponding to a function symbol f of arity n is the function that maps n terms $t_1, \ldots, t_n$ to the tree:



Let $\tau(t)$ denote the sort of a term t in the term algebra.

In abstract data type and program specification, the aim is to circumscribe the class of algebras that could be considered as an implementation of the types and operations defined. Although some of the function symbols appearing in a signature will be intended as uninterpreted 'constructor' functions, in general there will be certain relationships that must hold between the functions in any implementation of a particular data type. These relationships can be expressed as a set of equations involving terms from $T_\Sigma(X)$ that describe properties of the function symbols (e.g. the associativity, commutativity and idempotence — i.e. satisfying an equation of the form $f(x,x) = x$ — of the set union function) or define functions recursively or in terms of other functions, such as in the following definition for the list append function:

$$\begin{aligned} \text{append}(\text{nil}, L) &= L \\ \text{append}(\text{cons}(A, L_1), L_2) &= \text{cons}(A, \text{append}(L_1, L_2)) \end{aligned}$$

A Σ-*equation* is a triple $\langle X, t_1, t_2 \rangle$ where X is a finite S-indexed set of variable symbols, and t_1 and t_2 are terms of the same sort built from function symbols in Σ and variables from X. We will usually write an equation in the form

$$(\forall x_1 \colon s_1) \ldots (\forall x_n \colon s_n)\ t_1 = t_2$$

where $\bigcup_s X_s = \{x_1, \ldots, x_n\}$ and $x_i \in X_{s_i}$. The explicit declaration of the variables and their sorts is necessary to ensure the soundness of many-sorted equational deduction [Goguen & Meseguer 81, Meseguer & Goguen 85 (Section 4.3)]. We will omit these declarations where the sorts of variables are clear from context, and simply write $t_1 = t_2$.

Given a set of Σ-equations E, a (Σ, E)-algebra $\mathcal{A}$ is a Σ-algebra which *satisfies* each equation in E, i.e. for each equation $\langle X, t_1, t_2 \rangle \in E$ and each assignment $f \colon X \to \mathcal{A}$ of elements of the algebra to the variables of X, the equality $f^\#(t_1) = f^\#(t_2)$ holds in $\mathcal{A}$,

where $f^{\#}$ is the unique extension of f to a homomorphism mapping terms to elements of $\mathcal{A}$.

6.2.1 Initial Algebra Semantics

Specifying a signature and a set of equations is not sufficient to define a data type completely. For instance, an equational specification for the natural numbers will also be satisfied by the real numbers; and in general, there will be an infinite number of non-isomorphic algebras satisfying any specification. However, there is a well-defined notion of a 'standard' model which can be characterised by the following two properties (quoted from [Meseguer & Goguen 85]):

1. *No Junk*: Every data item can be constructed using only the constants and operations in the signature. (A data item that cannot be so constructed is 'junk').

2. *No Confusion*: Two data items are equivalent if and only if they can be proved equal using the equations. (Two data items that are equivalent but cannot be proved so from the given equations are said to be 'confused').

For a given signature, these conditions define an algebra that is unique (up to renaming of the elements), called the *initial* algebra (using the terminology of category theory [MacLane 71] — it is initial in the sense that there is a unique homomorphism from it to any other (Σ, E)-algebra). The 'no junk' condition is equivalent to the principle of *structural induction* [Burstall 69], whereby a property P can be proved for all data objects by showing it holds for all constants and also for all objects denoted by a term $f(t_1, \ldots, t_n)$ if P holds for $t_1, \ldots, t_n$.

The initial algebra can also be characterised as the quotient of the algebra of ground terms by the *equational theory* generated by the equations. For a set of equations E, an equational theory is the finest equivalence relation $=_E$ on terms in $T_\Sigma(X)$ generated by the following three rules:

1. $s =_E t$ if there is an equation $\langle X, s, t \rangle$ in E.

2. $=_E$ is a *congruence relation*, i.e. an equivalence relation with the additional property:

$$\bigwedge_{1 \le i \le n} s_i =_E t_i \Rightarrow f(s_1, \ldots, s_n) =_E f(t_1, \ldots, t_n) \quad \text{if } f \text{ has arity } \tau(s_1) \ldots \tau(s_n)$$

3. $=_E$ is closed under instantiation: if $s =_E t$ then $\sigma s =_E \sigma t$ for any substitution σ.

Thus the members of the initial algebra represent equivalence classes, where two terms are in the same class if and only if they can be shown to be equal by equational reasoning from the equations E. E is said to be an *equational presentation* of the theory.

In many cases, the equations in a theory presentation can be oriented and used as *rewrite rules*, and this provides a powerful mechanism for computing with terms in an equational theory. A rewrite rule is a directed equation $\ell \Rightarrow r$ where $vars(r) \subseteq vars(\ell)$. If ℓ matches some subterm s of a term t (so that $s = \theta\ell$ for some θ), then t can be 'rewritten' to produce the term t' that results from replacing $\theta\ell$ with θr in t. This computation step is called a *one step rewriting*, and it generates a relation $\rightarrow$ on terms: $t \rightarrow t'$ holds whenever some rule can be applied to rewrite t to t'.

If a set of rewrite rules satisfies certain properties, then each term will have a unique *canonical* or *normal* form representing its coincidence class in the corresponding equational theory. A sufficient condition is that the rules are *confluent* and *terminating*.

Let $\stackrel{*}{\rightarrow}$ denote the transitive-reflexive closure of the one step rewriting relation $\rightarrow$, i.e. $t_1 \stackrel{*}{\rightarrow} t_2$ states that t_2 can be generated from t_1 by a (possibly empty) sequence of applications of rewrite rules. A set of rewrite rules is confluent if whenever $s \stackrel{*}{\rightarrow} t_1$ and $s \stackrel{*}{\rightarrow} t_2$, there exists a term u such that $t_1 \stackrel{*}{\rightarrow} u$ and $t_2 \stackrel{*}{\rightarrow} u$. The termination condition holds if there is no infinite sequence of rewritings possible starting from any term. If these conditions hold then equality in the initial algebra can be decided by rewriting (ground) terms and testing for identity of their canonical forms. In fact, it is known that any computable data type (i.e. an algebra whose carrier sets are recursive sets and operations are recursive functions) can be described as an initial (Σ, E)-algebra for some signature Σ including the function symbols of the data type, and some confluent and terminating set R of rewrite rules (this result, due to Bergstra and Tucker, is discussed at length in [Meseguer & Goguen 85]). Rewriting can therefore provide

an operational semantics for abstract data types, with the evaluation of an expression corresponding to rewriting it to its canonical form.

Determining if a set of rewrite rules is terminating is an undecidable problem, as is testing for confluence in general. However, confluence is decidable for terminating sets of rewrite rules. Also, it may be possible to extend a terminating but non-confluent set of rules to a confluent set using the Knuth-Bendix completion procedure [Knuth & Bendix 70], although the algorithm can terminate signalling failure.

The use of initial algebras for the specification of abstract data types is discussed at length in [Meseguer & Goguen 85]. For more detail on rewrite rules see [Bundy 83] and [Huet & Oppen 80].

6.2.2 Conditional Equations and Rewrite Rules

Although equational specification together with initial algebra semantics can be used to define any computable data type, it may be necessary to add extra additional 'hidden' functions. Some data types are more naturally and conveniently expressed by the use of *conditional equations*. A conditional equation has the form $(\forall X)\, t_1 = t_2 \; if \; C$ where X is a set of variable symbols and C is a conjunction of unquantified equations with variables in X.

The extra expressive power of conditional equations is illustrated by Figure 6.1 (example from [Zhang 84], presented in [Bockmayr 86]), giving a specification for the integers with the function $ispos\colon int \rightarrow bool$ defined to have value $true$ for positive integers and *false* otherwise. For the given signature, there is no finite set of unconditional equations containing the equations $s(p(x)) = x$ and $p(s(x)) = x$ for which the desired algebra is initial.

A conditional equation may be oriented and used as a rewrite rule if the variables occurring in C and t_2 are a subset of those in t_1. A conditional rewrite rule $(\forall X)\, \ell \Rightarrow r \; if \; t_1 = u_1 \wedge \ldots \wedge t_n = u_n$ can be applied to a term t to replace a subterm $\sigma\ell$ with σr if $\sigma t_i = \sigma u_i$ for each i. The condition is tested by rewriting each σt_i and σu_i to their canonical forms and testing for equality (assuming the set of conditional rewrite rules is confluent and terminating). This recursive application of rewriting makes the rewriting relation $\rightarrow$ undecidable, as even a single application of a rewrite rule may lead

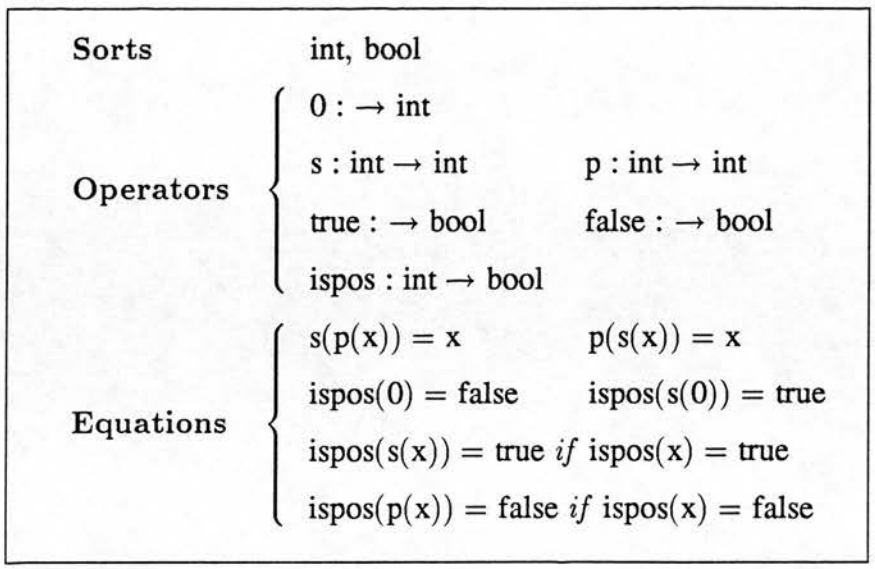


Figure 6.1: A specification using conditional equations

to an infinite loop. [Bockmayr 86] discusses a class of conditional rewriting systems due to Kaplan (1984) which are terminating and have a Knuth-Bendix style completion procedure.

6.2.3 Unification in Equational Theories

The discussion of rewrite rules above applies to the problem of deciding equality of ground terms in an equational theory. If a problem involves solving equations containing variables, as in logic programming, then *equational unification* is required. Given an equational theory T, and two terms s and t, equational unification is the problem of finding bindings for the variables occurring in s and t that will make these two terms equal. The usual notion of substitution extends to many-sorted and order-sorted algebra: substitutions must be well-sorted, mapping a variable of a sort S to a term of that sort. If σ is a substitution such that $\sigma s =_T \sigma t$ then σ is said to be a T-unifier of s and t.

Solving a unification problem in an equational theory T is much more difficult than standard unification. T-unification may not be decidable, and even if it is, it may be necessary to find multiple, or even an infinite number of unifiers.

Let $\prec_T$ denote the subsumption partial ordering on the set of all substitutions $\mathcal{S}$ in the

theory T: a substitution τ is more general than σ, written $\sigma \prec_T \tau$, if $(\exists\theta \in \mathcal{S})\ \sigma = \theta\tau$. Let the set of all T-unifiers of two terms s and t be denoted $U_T(s,t)$. Then a set of *most general unifiers* of two terms s and t is a subset Σ of $U_T(s,t)$ that satisfies the following conditions:

- Σ is *complete*: $\forall\sigma \in U_T(s,t)\ \exists\tau \in \Sigma\ \ \sigma \prec_T \tau$
- Σ is *minimal*: $\forall\sigma,\tau \in \Sigma\ \ \sigma \prec_T \tau \Rightarrow \sigma =_T \tau$

A theory is said to be *infinitary* if there exist a pair of terms with an infinite set of most general unifiers. If all unifiable pairs of terms have a finite number of most general unifiers then the theory is said to be *finitary*.

For some theories there are pairs of terms for which such minimal complete sets of unifiers do not exist. Even for theories where it is always possible to find a set of most general unifiers, this may involve filtering the output of the unification algorithm to remove redundant substitutions. Therefore, it is sometimes better to drop the minimality condition and simply find a *complete set of unifiers*.

6.2.3.1 Combining Unification Algorithms

Although unification algorithms are known for a number of specialised theories, such as various combinations of associativity, commutativity and distributivity axioms, there is no general method for extending or combining them if new equations are added. Even adding new uninterpreted function symbols to a theory may require finding a new unification algorithm. However, if a theory can be decomposed into a number of subtheories with no function symbols in common, then under certain limitations on the types of equations appearing, the separate unification algorithms can be combined in a straightforward way.

Yelick's *CR-unify* procedure [Yelick 85a, Yelick 85b, Yelick 87] applies to the case when the equations in the presentation of each subtheory are *collapse-free* (called *confining* by Yelick) and *regular* (hence the name).

An equation is a collapse axiom if it has the form $x = t$ or $t = x$ where x is a variable and t is a non-variable term, otherwise it is collapse-free. This restriction, together with the disjointness constraint between the sets of function symbols for each

subtheory, ensures that two terms can never be equal if their head symbols (i.e. principal functors) belong to different subtheories.

An equation $s = t$ is regular if the sets of variable symbols occurring in s and t are the same. The algorithm involves 'homogenising' the terms to be unified, i.e. substituting new variables for subterms whose head symbols belong to different subtheories from that of the term's principal functor. The two substitutions that recover the original terms by instantiating these variables are known as *preserving substitutions*. Regularity combined with the collapse-free condition ensures completeness of the algorithm by guaranteeing that a variable is never unifiable with a term containing that variable 'beneath' the homogeneous part of that term, i.e. in a subterm whose head symbol is from a different subtheory from that of the term's head symbol. Therefore, the algorithm can check for this case and fail without losing completeness (this check is needed to ensure termination).

The basic idea of the algorithm is to unify the homogenised terms in the appropriate subtheory and then combine each resulting unifier σ with the union of the preserving substitutions θ using a recursive process for unifying substitutions: choose $v \in Dom(\sigma \cup \theta)$, find all unifiers ϕ_i of the two bindings σv and θv for v and return the set of all substitutions $\psi_j \circ \phi_i$ where ψ_j unifies the substitutions $\phi_i \circ \sigma$ and $\phi_i \circ \theta$ (this description follows the presentation of Zachary (1987) rather than Yelick's iterative form of the algorithm). There may be many different ways to unify two substitutions, and to cope with infinitary equational theories (where two terms may have an infinite number of unifiers) the algorithm can be modified to produce a stream of substitutions. As Yelick notes, the overhead of forming the homogeneous terms can be eliminated by implementing the theory-specific unification procedures to operate directly on inhomogeneous terms, calling on the general top-level unification procedure to unify subterms whose head symbols belong to other theories.

Example If T_A and T_B are theories containing the left commutative bag insert operation $ins\colon elt \times bag \rightarrow bag$ and the uninterpreted function symbols $a, b\colon \rightarrow elt$ and $nil\colon \rightarrow bag$ respectively, then the solution of the unification problem $ins(a, x) = ins(y, ins(b, nil))$ proceeds as follows: First, the terms are homogenised to give

$ins(u, x)$ and $ins(y, ins(v, w))$ with preserving substitutions $\theta_1 = \{a/u\}$ and $\theta_2 = \{b/v, nil/w\}$. Let $\theta = \theta_1 \cup \theta_2 = \{a/u, b/v, nil/w\}$. The homogenised terms are unified in T_A (taking the left commutativity of ins into account) giving two unifiers $\sigma = \{y/u, ins(v, w)/x\}$ and $\tau = \{v/u, ins(y, w)/x\}$. τ cannot be unified with θ as this requires $a = u = v = b$. To unify σ and θ, we first unify the bindings of u to get $\phi = \{a/y\}$. Unifying $\phi \circ \sigma = \{ins(y, w)/x\}$ and $\phi \circ \theta = \{b/v, nil/w\}$ gives $\{ins(b, nil)/x\}$ (skipping the details), and the final result is the single substitution $\{ins(b, nil)/x\} \circ \phi = \{ins(b, nil)/x, a/y\}$.

Appendix C describes this algorithm diagrammatically using the language of category theory, and this motivates the development of a similar procedure for combining matching algorithms.

Other algorithms for combining unification algorithms were developed independently by Tidén (1986) (similar to Yelick's approach, but without the restriction to regular theories — although Zachary (1987) claims this improvement is inefficient), Herold (1986) and Kirchner (1985). Schmidt-Schauß (1989b) gives a procedure for combining the unification algorithms of a combination of arbitrary disjoint theories, although this has a high complexity.

6.2.3.2 Narrowing

An important problem in unification theory is finding algorithms which can be used for a whole class of unification problems, taking the pair of terms to be unified and an equational presentation of the theory as arguments, and producing a complete set of unifiers. This is called *universal unification.* For equational theories presented by a confluent and terminating set of rewrite rules, a complete set of unifiers can be computed by a technique called *narrowing*, although this set may be far from minimal in general.

Let T be an equational theory presented by a confluent and terminating set of rewrite rules R. Two terms s and t in $T_\Sigma(X)$ are T-unifiable if there is a substitution σ such that $\sigma s =_T \sigma t$. This is equivalent to

$$\mathrm{NF}(\sigma s) \equiv \mathrm{NF}(\sigma t) \qquad (*)$$

where NF(t) is the unique canonical form of t under R, and $\equiv$ denotes syntactic identity. If s and t are not unifiable by the standard unification algorithm, then at least one of the terms σs and σt must be reducible by R, or $(*)$ would not hold. This can be represented diagrammatically as follows:

$$\begin{array}{ccc} s \overset{\sigma}{\mapsto} \sigma s \to_R \cdots \to_R \text{NF}(\sigma s) & & \\ & \| & \\ t \overset{\sigma}{\mapsto} \sigma t \to_R \cdots \to_R \text{NF}(\sigma t) & & \end{array}$$

Narrowing provides a systematic method for finding all such T-unifiers σ by interleaving unification and rewriting steps. The basic narrowing step is to instantiate a term t sufficiently so that some subterm matches the left-hand side of a rewrite rule in R, which is then applied to produce a new term t'.

Let t_0 be a non-variable subterm of t and $\ell \Rightarrow r$ be a rewrite rule such that t_0 and ℓ are unifiable with most general unifier α (assuming that the variables have been standardised apart). Then if t' is the term obtained from αt by replacing the subterm $\alpha t_0 = \alpha \ell$ with αr, we say that t' is a *one step narrowing* of t with *narrowing substitution* α, and write $t \overset{\alpha}{\rightsquigarrow} t'$ or simply $t \rightsquigarrow t'$. The narrowing relation is the reflexive and transitive closure of one step narrowing. Note that rewriting is a special case of narrowing. The process of T-unifying two terms by narrowing can be represented by the following diagram:

$$\begin{array}{ccc} s = s_0 \overset{\alpha_1}{\rightsquigarrow} \cdots \overset{\alpha_n}{\rightsquigarrow} s_n \overset{\mu}{\mapsto} \mu s_n & & \\ & \| & \\ t = t_0 \overset{\alpha_1}{\rightsquigarrow} \cdots \overset{\alpha_n}{\rightsquigarrow} t_n \overset{\mu}{\mapsto} \mu t_n & & \end{array}$$

where μ is the most general unifier of s_n and t_n. The substitution $\sigma = \mu \circ \alpha_n \circ \cdots \circ \alpha_1$ is therefore a T-unifier for s and t. The T-unification algorithm given by Fay (1979) and extended by Hullot (1980), finds a complete set of T-unifiers by finding all such narrowing chains emitting from s and t.

The narrowing algorithm extends to equational theories defined by conditional rewrite rules [Bockmayr 86]. When applying a conditional rewrite rule, the condition may either be evaluated by recursively calling the narrowing algorithm, or it can simply be added to the new goal produced by the rewriting operation.

Narrowing (and rewriting) can also be used for theories where the equational presentation is partitioned into a set of rewrite rules R and a set of equations E that are built into the unification and matching algorithms. Computation proceeds as before, but with E-unification (or E-matching) used to unify (or match) the left hand side of a rule with the subterm being rewritten. The corresponding narrowing and rewriting relations are denoted by $\rightsquigarrow_{R,E}$ and $\rightarrow_{R,E}$. To use this form of narrowing, the confluence and termination conditions must hold 'modulo E', i.e. the relation $\rightarrow_{R,E}$ must be terminating, and whenever a term t reduces under $\rightarrow_{R,E}$ to two different terms t_1 and t_2, there must exist terms t_1' and t_2' such that $t_1 \stackrel{*}{\rightarrow}_{R,E} t_1'$, $t_2 \stackrel{*}{\rightarrow}_{R,E} t_2'$ and $t_1' =_E t_2'$. In fact, a stronger condition than confluence must hold. R should be *Church-Rosser* modulo E: if two terms t_1 and t_2 are equal in the equational theory $R \cup E$ (disregarding the orientation of the rewrite rules), they must reduce under sequences of $\rightarrow_{R,E}$ rewritings to give terms t_1' and t_2' such that $t_1' =_E t_2'$. If E is empty, this condition is equivalent to confluence, but this is not true in general [Jouannaud, Kirchner & Kirchner 83].

The equational logic programming language Eqlog [Goguen & Meseguer 86] is based on this extension of narrowing; and in general, narrowing is considered to be a promising framework for integrating functional and logic programming once appropriate techniques have been found to increase its efficiency.

A general survey of the theory and results of the study of unification is given in [Siekmann 89].

6.2.4 Order-Sorted Algebra

Using many-sorted algebra for the specification of abstract data types provides a strong-typing discipline and helps ensure that meaningless expressions are not included in the algebra. However, this can be too restrictive for many data types. A data type specification may involve sorts that are contained within other sorts, i.e. all objects of a sort S_1 may also be objects of another sort S_2. S_1 is said to be a *subsort* of S_2, which is a *supersort* of S_1. In this case, there may be some overloaded function symbols, defined on the two sorts such that the function on the supersort is an extension of the equivalent function on the subsort. For example, '+' on natural numbers is the same operation as '+' on the integers restricted to the natural numbers. This type of

relationship between sorts and their operations cannot be naturally expressed in standard many-sorted algebra.

Order-sorted algebra (discussed in detail in [Goguen & Meseguer 87a]) extends many-sorted algebra by allowing a partial order to be defined on the sorts of a signature with subsorts inheriting operations from their supersorts. A term may then have more than one sort (e.g. '1' is both a natural number and an integer), but if the arities and sorts of the operators satisfy certain very natural conditions then every term has a unique *least sort* and the theory of many-sorted algebras extends in a straightforward way to order-sorted algebra. In particular, a specification in order-sorted equational logic has an initial model which can be expressed as a quotient of the term algebra by the equational theory. Also, a specification using order-sorted algebra can be translated to an equivalent specification using standard many-sorted algebra with additional 'coercion' functions which are injections from sorts to their supersorts, and extra equations describing how coercions interact with the other operators. This provides an efficient operational semantics for order-sorted algebra: terms are translated to their 'least parse' in the standard many-sorted algebra (with coercions) and then reduced (or narrowed) using the extended set of equations [Goguen, Jouannaud & Meseguer 85].

When subsorts are used, it is not always possible to determine if a term is well-formed while parsing it. For instance, although the term $(4/2)!$ will become well-formed after reduction, there is no way for the parser to determine that $4/2$ is a real number that also happens to be an integer. To avoid this problem, terms that are possibly ill-formed can be given the benefit of the doubt at parse time by allowing the parser to insert special operators called *retracts* which are left inverses to the coercion functions [Goguen, Jouannaud & Meseguer 85]. A retract can be used to fill in the gap between the least sort of a term (as determined by the parser) and a smaller sort that the parser requires that term to have in order to parse the enclosing term. If $\mathtt{r}_{\mathtt{Real,Int}}$ is a retract that maps between the sorts `Real` and `Int`, then $(4/2)!$ can be parsed as $\mathtt{r}_{\mathtt{Real,Int}}((4/2)!)$. During evaluation, this retract will cancel with a coercion operator and execution can proceed normally. If such a term does not become well-formed, the retract will remain unreduced to act as an informative error message.

6.2.4.1 Subsorts and Partial Functions

Data types often have operations which are not meaningful for some objects of that sort, for example, it should not be possible to find the tail of an empty list, or to pop a value off an empty stack. Using order-sorted algebra, a partial function on a given sort can be defined as a total function on the set of objects for which it is well-defined, e.g. the tail function on lists is total on the subsort of non-empty lists.

Partial functions can also be defined using error supersorts, which contain the well-defined objects as a subsort and possibly also contain special error values. For instance, we could define $tail(nil) = no\_tail$ where no_tail is an error value of sort err_list, a supersort of $list$.

Subsorts are defined by the constructor functions that generate the elements of that sort. For instance, non-empty lists are generated by the constructors $nil: \rightarrow list$ and $cons: list\_element \times list \rightarrow ne\_list$ while (possibly empty) lists can also be produced by the function $tail: ne\_list \rightarrow list$. However, the domains of partial functions cannot always be expressed in this way. *Sort constraints* [Goguen, Jouannaud & Meseguer 85] can be used to declare that the result of a function has a smaller sort than expected for all argument values satisfying a given condition. This can be used together with an error supersort to define more general types of partial function. For instance, the 'push' operation on a bounded stack of natural numbers can be declared in OBJ2 [Futatsugi et al. 84] by the following specification fragment:

```
sorts NeStack Stack ErrStack .
subsorts NeStack < Stack < ErrStack .
op push : Nat Stack -> ErrStack .
var N : Nat .
var S : Stack .
as NeStack : push(N,S) if length(S) < bound .
```

(assuming the data type `Nat` is already defined, and omitting declarations for `length`, `bound`, other stack constants and operations, and the equations). The "*as*" statement is a sort constraint declaring that `push(N,S)` is a non-empty stack when `length(S) < bound` (otherwise its least sort is `ErrStack`, the declared co-arity of `push`). If the operation `pop`

is defined for elements of `NeStack`, then the term `pop(push(N,S))` will be parsed as $\texttt{pop}(r_{\texttt{ErrStack,NeStack}}(\texttt{push(N,S)}))$ where $r_{\texttt{ErrStack,NeStack}}$ is a retract. If S becomes instantiated to a stack with length less than `bound`, then this retract will cancel with a coercion $c_{\texttt{NeStack,ErrStack}}$, otherwise it will remain to form a run-time error message.

6.2.5 Modularity

A data type can often be most naturally described as a structure built from a number of simpler data types, or as an extension of an existing type. Also, some types are generic by nature, and can be described independently of their 'argument' types as parameterised specifications. For instance, lists can be described using constructors *nil* and *cons* and functions *head*, *tail* and *append* satisfying particular equations, regardless of the type of the list elements. It is therefore important to support the specification of data types and computations in a modular fashion, maximising the reusability of components.

In the algebraic approach to specification, a data type is described by a theory and operations for combining modules correspond to functions on theories. In particular, a generic module can be expressed as a *theory procedure*, taking particular theories for the parameter types as arguments and producing a specialised theory as a result. The actual parameter theories given in any application of a theory procedure must usually satisfy certain requirements. For instance, a generic module for lists would be parameterised by a theory describing the data type of the list elements. The only requirement on this theory is that it has at least one sort. A theory procedure for lists with a sorting function defined also requires that the parameter theory is equipped with an ordering predicate (regarded as a boolean-valued function). Theory procedures express these constraints by declaring *requirement theories* as formal parameters. An instance of a generic module is constructed by providing the argument theories together with functions that map from the requirement theories to the actual parameter theories, describing which sorts and functions correspond to those named in the requirement theory. These *views* (or *fitting morphisms*) must be homomorphisms between theories and this ensures that the requirements of the theory procedure are satisfied. The requirement theories of a generic module are interpreted *loosely*—any algebra satisfying the specification is a

possible candidate for that parameter.

The sorts and operators introduced in a theory procedure may be intended to have a standard interpretation, i.e. any model of the generic specification should satisfy the 'no junk' and 'no confusion' properties with respect to these sorts and operators. However, we want to allow any interpretations for the parameters' requirement theories. This notion is captured by the idea of a *free extension.* See [Goguen & Meseguer 87a] for a discussion of this in a particular logic; [Goguen & Burstall 83] for the general case.

Theories can be *enriched* by adding new sorts, operators and equations to a previously defined theory. This can change the initial model of the subtheory. For instance, enriching a specification for lists by adding the left-commutativity axiom

$$\mathrm{cons}(\mathrm{A}, \mathrm{cons}(\mathrm{B}, \mathrm{L})) = \mathrm{cons}(\mathrm{B}, \mathrm{cons}(\mathrm{A}, \mathrm{L}))$$

produces a specification for bags. It is useful to be able to constrain the interpretation of a subtheory so that this cannot happen, e.g. to prevent future enrichments changing that part of the specification, or to assert in a requirement theory that the corresponding parameter theory should include a certain data-type in its standard interpretation. In this case, the enrichment should be a *protecting* extension [Goguen & Meseguer 87a], i.e. the extended theory, restricted to the sorts and operators of the old theory, should be isomorphic to the old theory. This is particularly important when part of a module is implemented by a built-in data type — the implementation will still be correct if this condition holds.

It is important to have a clearly defined semantics for operations that combine theories, so that there is no ambiguity about the interpretation of the objects of the combined theory. For example, there should be an intuitive answer to the question of whether sorts with the same name in two different theories will denote a single sort when the theories are combined. The program specification language Clear [Burstall & Goguen 81] and the programming languages OBJ2 and Eqlog provide operations on theories based on concepts from category theory [Pierce 90, Goldblatt 79, Arbib & Manes 75, MacLane 71]. When a theory is constructed, any *named* theories that form a part of it are recorded as ancestors of that theory. In the combination of two theories, any sorts and operators of the component theories that 'coincidentally' have

the same name are regarded as distinct, i.e. the new theory has the disjoint union of the sorts and operators of the components, except for those occurring in any common ancestor theories which are not duplicated. Also, each application of a theory procedure generates new instances of the sorts and operators of the procedure body. For example, the combination of modules specifying lists of integers and lists of characters (denoted $List(Int) + List(Char)$) would contain two sorts called *list*, two instances of the constructors *nil* and *cons*, and two functions called *append*, etc. $List(Int) + List(Int)$ would be similar, but would only contain one sort called *int*. The semantics underlying this uses the notion of *based theories* (represented as *cones* in category theory) with the combination of two theories defined as their *coproduct*. See [Burstall & Goguen 80] or [Sannella 82] for details.

6.2.6 Existing Systems

This section gives a brief description of some of the specification and programming languages that use the techniques described above.

Clear [Burstall & Goguen 81, Burstall & Goguen 80] is a language for specifying programs using the algebraic approach. It is designed for generality, allowing any logical system to be used for the axioms of a theory, provided it satisfies certain conditions that make it appropriate for specification. Such a system is called an *institution* [Goguen & Burstall 83]. Clear provides operations for combining and enriching theories, forgetting and renaming sorts or operators, and parameterised theories. Initial models and free extensions are declared using *data constraints* indicating that particular subtheories have a standard interpretation. It is implemented directly using constructs from its category theory semantics (see [Rydeheard & Burstall 88] for a discussion of this approach), although there is also a more efficient implementation based on a more intuitive but less general semantics [Sannella 82].

OBJ2 [Futatsugi et al. 84] is a functional programming language based on order-sorted equational logic. Abstract data types are defined by equational theories (using initial algebra semantics), presented in the form of a set of conditional rewrite rules together with a set of equational 'attribute' declarations for particular function symbols. For each function symbol, these attributes describe which of a number of 'built-in'

equational axioms it satisfies: associativity, commutativity, idempotence (i.e. satisfying an equation of the form $f(x,x) = x$), having a specified constant as an identity, or any combination of these. The attributes are used to select the appropriate pattern matching algorithms to use during each step of rewriting, and to normalize the rewritten term with respect to these properties. The commutativity attribute is implemented simply by sorting the arguments in any nested series of applications of a commutative operator. Therefore, it is sometimes necessary to give two versions of rewrite rules whose left hand sides involve commutative operators with partially uninstantiated arguments; e.g. both of the equations $S \cap \varnothing = \varnothing$ and $\varnothing \cap S = \varnothing$ are needed in a specification of sets — the rule that is used will depend on the relative ordering of $\varnothing$ and the term that matches S.

The user must ensure that the rewrite rules are terminating and Church-Rosser modulo the declared built-in axioms, however "the problem is not worse than that of avoiding non-termination in standard Prolog, since the programmer is likely to have a good intention of what he writes" [Goguen & Meseguer 86] and in fact "experienced programmers usually write rules that satisfy these properties" [Futatsugi et al. 84].

The module system in OBJ2 allows generic data types which are defined using parameterised theories, and a module may include other modules as subtheories with three types of declaration: *using*, where the imported module is simply copied; *protecting*, which asserts that the subtheory cannot be changed in the new module by generating new objects of imported sorts (creating 'junk') or by adding new equations affecting the subtheory (causing 'confusion'); and *extending* — asserting an easily checked condition guaranteeing 'no confusion'.

Eqlog [Goguen & Meseguer 86] is a language unifying the features of logic and functional programming by extending the capabilities of OBJ2 to allow the solution of goals containing variables. This is done by a combination of Prolog-style computation and narrowing. The underlying logic is order-sorted Horn clause logic with equality [Smolka 86], i.e. the signatures of modules can introduce predicate symbols as well as sorts and function symbols, and the functions are defined using Horn clauses which may involve a distinguished equality predicate. **FOOPlog** [Goguen & Meseguer 87b] is a proposed language to extend these ideas even further, combining features of functional,

relational and object-oriented programming.

TEL [Smolka 87] is a language integrating relational programming (using Horn clauses) with functional programming (using conditional equations as rewrite rules). It has a powerful sort system with subsorts and polymorphic sort constructors (allowing parametric polymorphism, as in ML).

6.3 Multi-Valued Mode Systems

Multi-valued modes were introduced by Zachary (1987, 1988) as part of a methodology for incorporating data abstraction principles into a Prolog-like language. In the language Denali, each data type to be used in the program is defined using an equational logic specification (called a data-abstraction in Zachary's terminology). A data-abstraction introduces a new sort name, declares the functions that produce objects of that sort, and presents a set of equations that define when two syntactically different terms of that sort should be considered equal. For instance, Zachary's specification for natural numbers includes the components shown in Figure 6.2.

Sort	nat
Operators	$0 : \rightarrow \text{nat}$
	$s : \text{nat} \rightarrow \text{nat}$
	$+ : \text{nat} \times \text{nat} \rightarrow \text{nat}$
Equations	$X + 0 = X$
	$X + Y = Y + X$
	$s(X) + Y = s(X + Y)$

Figure 6.2: A specification for natural numbers

As Denali is a resolution-based language like Prolog, the equations are not used to evaluate functions by rewriting terms, as in OBJ2, or to solve equations by equational unification together with narrowing and rewriting as in Eqlog. Instead, they are used exclusively in the unification procedure of Denali. Zachary claims that narrowing is not yet sufficiently understood to provide a practical and efficient method for implementing

unification in arbitrary equational theories. His approach is to provide support for the user to define sort-specific unification predicates for each data abstraction; although algorithms for some common theories, such as unification under associativity and commutativity, could be built in to an implementation.

Designing equational unification algorithms is a non-trivial task, and there are a number of problems that the user may confront. The problem of whether two arbitrary terms in the theory are unifiable may be undecidable. This can be the case even for simple combinations of equations (e.g. unifying in a theory with two function symbols and distributivity and associativity axioms [Siekmann 89]). Even if the unification problem is decidable, finding a complete set of unifiers may be very expensive and possibly intractable.

To help the user overcome these difficulties, Zachary notes that the complexity of finding complete sets of unifiers is reduced if the unification procedure only needs to operate over a subset of the terms of that sort, and he provides a mechanism for specifying a very natural form of restriction on the pairs of terms that can be unified. This is done by generalising the mode system first introduced by Warren to increase the efficiency of compiled Prolog programs [Warren 77].

6.3.1 Controlling Inference using Modes

Warren suggested annotating predicate definitions with *mode declarations* stating the intended usage of the arguments of a predicate. Although Prolog procedures have no inherent 'directionality' — the same predicate can be used to instantiate variables occurring in different arguments on different occasions — the programmer may know that certain arguments are guaranteed to be instantiated when the procedure is called (therefore acting as 'inputs') and other arguments will become instantiated as a result of the query (thus acting as 'outputs'). Declaring when this is the case can allow a Prolog compiler to optimize the compiled code.

Mode distinctions of this type are also used in some extensions of Prolog to declare that procedure calls for a particular predicate should be delayed until the arguments are sufficiently instantiated, thereby helping to avoid non-terminating computations. For instance, the *freeze* declaration of Prolog-II [Colmerauer 82] delays a subgoal until

a particular variable is bound to a non-variable. The *wait* declaration of Mu-Prolog [Naish 85] allows the user to specify that calls to a particular predicate should be delayed until certain prespecified arguments are at least as instantiated as the corresponding arguments in the clause head. This prevents goal variables occurring in these argument positions from being instantiated during unification with the clause head, which could lead to an infinite loop when recursive calls are made.

Concurrent logic programming languages use similar techniques to synchronise the use of variables shared by concurrent processes.

Because the important distinction in these systems is whether certain terms are variable or non-variable, Zachary calls this type of mechanism a *bi-valued* mode system. In Denali, the computation (including unification) can be controlled using *multi-valued modes* which can describe the *degree* of instantiation of a term by taking its internal structure into account. The benefits of this are illustrated by considering the unification of the terms $X + Y$ and $Z + W$ of sort *nat*. A complete unification algorithm must return an infinite set of unifiers. However, if it is guaranteed that no *nat* term presented to the unification procedure will ever contain more than one variable, then the algorithm is only required to look for one most general unifier.

6.3.2 Multi-Valued Modes

In logic programming, computation takes place over a domain of terms built from variables and the function symbols of the underlying logical theory. The set of all terms that are well-formed, i.e. those that respect the declared arities and sorts of the function symbols, is called the *term algebra*. With a multi-valued mode system, computation can be controlled by restricting the inputs of certain procedures to the subset of terms that satisfy a particular property. This is done by declaring for each sort a number of *modes*.

A mode for a sort S is a set of terms of sort S which have a common property such as 'containing no variables' or 'having no occurrence of the function symbol "$\cup$" '. As deciding general properties of terms is not practical, or even possible[2], multi-valued

[2]Via Gödel numbering, the halting problem can be expressed as a test for membership of a mode of sort *nat*.

modes are defined in terms of the *structural* properties of their members. For technical reasons, some restrictions (outlined below) must be placed on the sets of terms that can form modes. Interested readers should consult [Zachary 87] for details.

A term t is said to have mode M, written $M(t)$, if $t \in M$. If a term has mode M, for some M, then it is *moded*, otherwise it is *unmoded*. Note that a term may have more than one mode.

In Denali, a set of mode n-tuples $\{\overline{M}_1, \ldots, \overline{M}_m\}$ is associated with each n-ary predicate symbol P. An atomic formula $P(t_1, \ldots, t_n)$ is *well-moded* with respect to a mode tuple $(M_1, \ldots, M_n)$ if each t_i has mode M_i. The *moding* of P is the set of all term tuples $(t_1, \ldots, t_n)$ such that $P(t_1, \ldots, t_n)$ is well-moded with respect to one of the mode tuples $\overline{M}_j$ associated with P.

The modings of predicate symbols are used to delay the evaluation of goals that are not sufficiently instantiated. Only goals that are well-moded can be selected for evaluation. Denali also allows predicates to be defined by separate clauses for different modes of use. This is done by grouping the alternative clauses into a *guarded block*, in which each clause is preceded by a mode tuple (called a *mode guard*). These mode guards are used to select the appropriate clause to use based on the argument modes of the literal being reduced.

It is important to ensure that a goal that is well-moded will remain well-moded after other goals have been satisfied. This is guaranteed by identifying a subset of the term algebra called the *moded base*, containing all variables and all moded terms, and insisting that this set is closed under unification and the application of *moded substitutions* — those substitutions that map variables to moded terms. The moded base is specified by defining for each sort S a distinguished mode called any which contains all variables and any non-variable terms of sort S that are declared by the user to have mode any or any other more specialised mode of that sort. Thus, for a given sort, all modes are subsets of the mode any. The moded base is then defined to be the union of the modes any for all sorts. With this scheme, variables are always moded, but to ensure that modes are closed under moded substitution, no mode strictly smaller than any may contain variables.

Definitions (Zachary): A *moded base* for an equational theory E is any set of terms that contains all of the variable terms and is closed under moded instantiation, equality, and unification.

- (instantiation) $moded(\sigma) \wedge moded(t) \Rightarrow moded(\sigma t)$
- (equality) $moded(r) \wedge r =_E t \Rightarrow moded(t)$
- (unification) $moded(r) \wedge moded(t) \Rightarrow$
  $\exists\, \Sigma$ s.t. Σ is a complete set of E-unifiers of r and t,
  and $\sigma \in \Sigma \Rightarrow moded(\sigma)$

A *mode* M of sort S is any set of moded terms of sort S that is closed under moded instantiation and equality.

- (instantiation) $moded(\sigma) \wedge t \in M \Rightarrow (\sigma t) \in M$
- (equality) $r \in M \wedge r =_E t \Rightarrow t \in M$

6.3.3 Defining Modes

In a multi-valued mode system, a set of modes is defined for each sort by presenting a *mode signature*. This declares the mode names and contains a set of declarations of the form $f: M_1 \times \cdots \times M_n \rightarrow M$ where f is a function symbol with rank $\langle (S_1, \ldots, S_n), S \rangle$ and $M_1, \ldots, M_n$ and M are modes of sorts $S_1, \ldots, S_n$ and S respectively[3]. These declarations describe how the modes of a term depend on its principal functor and the modes of its subterms. The pair $\langle (M_1, \ldots, M_n), M \rangle$ will be referred to as a *mode rank* for f. When there is an inclusion relationship between modes of the same sort, the mode signature may be abbreviated by presenting this ordering in the mode signature. This is illustrated in Figure 6.3 which shows a mode signature for the sort *nat* (as defined in Figure 6.2). For this signature it was possible to omit declarations such as '0: $\rightarrow$ any' and 's : ground $\rightarrow$ any' as these can be inferred from the mode-inclusion ordering. With this mode signature, the term 's(0) + s(s(0))' has mode *ground*, 's(X) + 0' has mode *any* and 'X + s(Y)' is unmoded.

[3]This differs from Zachary's terminology where *mode signature* refers to the individual declarations. The terminology used here is chosen because of its correspondence with the use of *signature* and *rank* in Section 6.2.

modes: any, ground

submodes: any $>$ ground

$0 : \rightarrow$ ground

s : ground $\rightarrow$ ground

s : any $\rightarrow$ any

$+$: ground $\times$ ground $\rightarrow$ ground

$+$: ground $\times$ any $\rightarrow$ any

$+$: any $\times$ ground $\rightarrow$ any

Figure 6.3: A mode signature for the sort nat

A mode signature must always contain declarations for the mode any of that sort, and it should appear as a maximum element in the inclusion ordering.

Mode names may be overloaded between sorts. Where necessary, we will subscript modes with their sort name to avoid ambiguity.

It is possible that modes defined in this way do not satisfy the closure conditions presented above. There are two problems that could arise. First, a mode may not be closed under equality. To test for this, each equation $s = t$ appearing in a data type specification must be checked to ensure that s and t belong to the same sets of modes. This must be done for all possible assumptions about the modes that variable subterms may have after instantiation. The second potential problem is that the moded base may not be closed under unification. Zachary states that devising a test for this is an open problem.

6.3.4 Moded Equational Unification

In Denali, there are two ways in which the sort-specific unification procedures can be provided. First, the equational theory for a user-defined sort may be a commonly occurring theory such as the theory with one associative and commutative function symbol (e.g. a bag union operator). In this case, the required unification algorithm may be built-in and it is sufficient to describe the theory required. Such a sort is said

to be *implicitly implemented.* If the unification theory of a sort is not provided in the language, the sort must be *explicitly implemented.* A translation predicate is given, describing how objects of that sort can be represented as objects of another, previously implemented sort. The unification procedure, possibly with mode restrictions on the terms to be unified, is defined as a predicate taking pairs of terms of the representation sort as arguments. For example, there is no known unification algorithm for the set insert function *insert* satisfying the following equations:

$$\text{insert}(X, \text{insert}(Y, S)) = \text{insert}(Y, \text{insert}(X, S))$$
$$\text{insert}(X, \text{insert}(X, S)) = \text{insert}(X, S)$$

Therefore, Zachary gives an implementation where sets are internally translated into objects of sort *bag* (which has constructors *nil* and *cons*) and are unified using the following predicate:

```
setUnify(nil,nil).
setUnify(cons(X,B1),cons(X,B2)) ← setUnify(B1,B2).
setUnify(cons(X,B1),cons(X,B2)) ← setUnify(cons(X,B1),B2).
setUnify(cons(X,B1),cons(X,B2)) ← setUnify(B1,cons(X,B2)).
```

When this predicate is used, unification will be done in the equational theory for *bag* which contains a left-commutativity axiom for `cons` (see Figure 6.4).

The mode restrictions on unification predicates are given by declaring a particular unification mode for the sort. At least one of the terms to be unified must be of the specified mode; the other term must be moded (i.e. of mode *any*). If a sort S has unification mode U, then the unification predicate for S has moding $\{(any, U), (U, any)\}$.

For example, Zachary restricts the `setUnify` predicate to the moding $\{(any, any)\}$, where the set mode *any* is defined by the declaration '`any` *from* `enum`'. This declares the mode *any* to contain all sets whose representation is a bag of mode *enum* — a 'fully enumerated' bag which has no variables of sort *bag*. By definition, the mode *any* must also contain all set variables. Therefore, the unification literal `setUnify(X,cons(Y,nil))` is well-moded but `setUnify(cons(1,X),cons(Y,B))` is not. This mode restriction is summarised by declaring the set unification mode to be *any*.

Mode restrictions can also be given for the unification procedures of implicitly implemented sorts.

The sort-specific unification procedures should be *modally complete*. A unification procedure is modally complete if it is complete for all pairs of terms from the moded base and it only produces moded substitutions. This prevents well-moded terms becoming ill-moded through unification.

An example of the use of modes in a specification is given in Figure 6.4. This shows Zachary's implicit implementation for bags of natural numbers (omitting the predicate definitions for associated operations such as *length*). This assumes there is a built-in algorithm for unification under left-commutativity. The presentation here does not use the syntax of Denali's module scheme.

Sorts	bag
Operators	nil : $\rightarrow$ bag cons : nat $\times$ bag $\rightarrow$ bag
Equations	$\text{cons}(X, \text{cons}(Y, B)) = \text{cons}(Y, \text{cons}(X, B))$
Modes	any $>$ enum $>$ ground nil : $\rightarrow$ ground cons : ground $\times$ ground $\rightarrow$ ground cons : any $\times$ enum $\rightarrow$ enum cons : any $\times$ any $\rightarrow$ any
Unification Mode	enum

Figure 6.4: A moded specification for bags of natural numbers, assuming sort *nat* is already defined.

6.3.4.1 Combining Moded Unification Procedures

Unifying two terms in Denali may involve solving subsidiary unification problems for subterms of various sorts with differing equational theories and mode restrictions.

To combine the unification algorithms for the different sorts, Zachary uses an extended version of Yelick's combining procedure *CR-unify* (see Section 6.2.3.1), so

named because of the restriction to confined (collapse-free) and regular theories. The theory being specified is partitioned into subtheories, one for each sort, each with its own unification algorithm implemented either implicitly or explicitly. The restriction to collapse-free theories can be eliminated because it is safe to assume that terms whose head symbols lie in different partitions are not unifiable. In the unsorted case this assumption could lead to incompleteness, but when the subtheories are partitioned along sort boundaries, such terms have incompatible sorts and no potential unifiers are lost. It is also possible to weaken the regularity condition on equations to permit equations that are *sort-regular*, a condition that depends on the ranks of the function symbols involved.

Three of Zachary's extensions are straightforward: adapting the procedure to many-sorted theories, catering for infinitary equational theories by returning (possibly infinite) streams of unifiers, and including self-contained unification algorithms for explicitly implemented sorts. The other extension involves dealing with the mode restrictions of the sort-specific unification procedures.

Let t_1 and t_2 be two terms of a sort S which has unification mode U. If neither term has mode U a mode failure should be reported; otherwise *CR-unify* must find a complete set of unifiers for t_1 and t_2. However, satisfying the mode restrictions for sort S does not guarantee that the unification can be successfully completed. Unifying t_1 and t_2 involves homogenising the terms and sending the subterms that are not of sort S to the unification procedures for the appropriate subtheories. Because the subsidiary unification algorithms are user-defined, they could have incompatible mode restrictions and therefore the combining procedure must delay any ill-moded subsidiary unification goals until they become further instantiated, signalling a mode failure if no further progress can be made.

It is also important to ensure that the combined unification algorithm can never generate an unmoded substitution. This is guaranteed as *CR-unify* generates unifiers by combining the substitutions produced by the separate unification procedures. If these are all modally complete then so is the combined procedure.

6.3.5 Extending Moded Unification to Order-Sorted Logic

Denali does not support the use of subsorts in specifications. Here we outline a simple extension to the multi-valued moding system to allow its use with order-sorted algebra.

6.3.5.1 The Equality Closure Anomaly

Recall that a mode M of a sort S is a set of moded terms of sort S that is closed under moded instantiation and equality. For order-sorted logic the equality closure condition is too strict as a term may be equal to another term whose least sort is greater. For example, suppose we have a module defining the integers (sort int) with non-zero integers (nz_int) as a subsort and the mode $ground$ defined in the obvious way. We would like this mode to remain well-defined regardless of what other data types are specified (provided that the addition of the new declarations is a *protecting* extension, i.e. no new objects of existing sorts are generated and there are no new equations identifying previously distinct objects). However, $ground$ will no longer be closed under equality if it is imported into a specification for rational numbers (rat) that includes the following components:

$$\text{int} < \text{rat}$$
$$/ : \text{int} \times \text{nz\_int} \rightarrow \text{rat}$$
$$(\forall \text{N}: \text{int})\ \text{N}/1 = \text{N}$$

The mode $ground$ must now contain the term '$1/1$' as $1/1 = 1$ in the equational theory for rat. This is not possible as $ground$ is a mode of sort int and the least sort of the '$1/1$' is rat.

We solve this problem by weakening the equality closure conditions for modes and the moded base that were presented in Section 6.3.2. Only equal terms whose least sort is the same or smaller should be considered:

- (equality closure for the moded base)

$$moded(r) \wedge r =_E t \wedge ls(t) \leq ls(r) \Rightarrow moded(t)$$

- (equality closure for modes)

$$r \in M \wedge r =_E t \wedge ls(t) \leq ls(r) \Rightarrow t \in M$$

where ls(t) is the least sort of a term t.

This means that a moded term may become ill-moded if any subterm is replaced with another equal term, but if this replacement is sort-decreasing the problem does not arise.

An alternative approach to weakening the equality condition would be to insist that any existing mode for a sort S must be extended to a mode for any sort $S' > S$ which has an equation $t = u$ (or $u = t$) such that $ls(t) \leq S < ls(u) \leq S'$. In the example above, a mode *ground* would have to be defined for the sort *rat* with $ground_{int} \subseteq ground_{rat}$.

If the equational theory is defined by a confluent and terminating set of rewrite rules R together with some 'built-in' equational axioms E, then we can weaken the equality closure condition even further. The modes and the moded base must be closed under the equality relation $=_E$, but it is only necessary for them to be closed under R-rewriting. This latter property will hold if each rewrite rule is *mode decreasing*: for all possible modes that the variable subterms may have after instantiation, whenever the left hand side is moded, every mode to which the right hand side belongs must be a submode of some mode of the left hand side. For conditional rules, this test should be restricted to only consider instantiations by substitutions that satisfy the rule's condition; however, this makes the test undecidable. For example, such a test could be used to solve Hilbert's tenth problem, which is known to be unsolvable [Davis 82]. Given a theory of integers (with the single mode *ground*) augmented with an extra, unmoded constant u, consider the rule $p \Rightarrow u$ if $p = 0$ where p is a polynomial with integer coefficients. This rule is mode decreasing precisely when $p = 0$ has a solution in integers.

In practical applications, the conditions of rewrite rules will always be satisfiable, and in many cases the conditions of the rules will not restrict the modes of the possible instantiations of variables in the rule, and the conditions can therefore be ignored for the purpose of this test. However, the resulting test is too strong: a conditional rule that fails may in fact be mode decreasing, and therefore this test can only be used to warn the user about possibly illegal rules. A closer approximation to the correct test could be achieved by considering all n-step narrowings of the rule's condition (for some small n). This would limit the possible bindings for the variables in the left hand side, and therefore restrict the modes that it may have.

6.3.5.2 Mode Inheritance

In Denali, a mode name must be unique for a given sort but may be overloaded between sorts. If S and S' are distinct sorts, each having a mode called m, then the two modes m_S and $m_{S'}$ have no connection. When subsorts are used, it is necessary to impose restrictions on the overloading of mode names between sorts and their subsorts.

First, we note that a mode defined on a sort can be used as a mode for its subsorts by restricting it to the smaller sort:

Proposition Let S and S' be sorts with $S < S'$. If M' is a mode of sort S' then the restriction $M'\restriction_S$ of M' to the elements of sort S is well-defined as a mode for S.

Proof

Closure under moded instantiation: If t has sort S then σt has sort S for any (well-sorted) substitution σ. This follows by induction on the structure of t. If t is a variable then σ can only instantiate it to a term of sort S. Otherwise, $t = f(t_1, \ldots, t_n)$ for some $n \geq 0$ where f is a function symbol with a declared rank $\langle (S_1, \ldots, S_n), S'' \rangle$ such that $S'' \leq S$. Now, $\sigma t = f(\sigma t_1, \ldots, \sigma t_n)$ and by the induction hypothesis σt_i has sort S_i for each i. Therefore, σt has sort S'' which is a subsort of S. We know that M' is closed under moded substitution, so if $t \in M'\restriction_S$ and σ is a moded substitution then $\sigma t \in M'$ and it follows that $\sigma t \in M'\restriction_S$.

Closure under equality: Suppose $t \in M'\restriction_S$ and $t =_E u$ where $ls(u) \leq ls(t)$. From the equality closure condition for mode M' we must have $u \in M'$. Now, $ls(t) \leq S$ as t has sort S. Therefore, $ls(u) \leq S$ and so u has sort S and must be in $M'\restriction_S$.

■

Thus there is a well-defined notion of inheritance whereby a sort can inherit modes from its supersorts. For example, a mode $enum$ for lists can be used as a mode for non-empty lists. Conversely, we may wish to extend a mode to a supersort by adding new mode rank declarations, e.g. in the rational numbers example we could define the mode $ground_{rat}$ by adding the submode constraint $ground_{int} < ground_{rat}$

and the declaration $/ : ground_{int} \times ground_{int} \rightarrow ground_{rat}$. These examples motivate the following conventions for using modes in order-sorted logic:

- A mode M for a sort S may be used as a mode for any subsort S'. This is taken to refer to the restriction of M to the sort S'.
- To avoid ambiguity, if a mode name M is overloaded between a sort S and a subsort S' then the condition $M_{S'} = M_S \restriction_{S'}$ must hold.

The second condition can be enforced by forbidding any mode M for a sort S to be declared if there already exists a mode M for a sort $S' > S$. No flexibility is lost, as the mode M of S' can be used as a mode for S. However, it is not possible to apply this restriction in the other direction: if a mode M exists for sort S, then it may be necessary to declare a mode M for a supersort S' in order to extend M to S'. In this case, however, it should be ensured that M of S' really is an extension of M of S. This can be done by asserting a submode constraint $(M \text{ of } S) < (M \text{ of } S')$ and making sure that none of the new mode ranks declared for mode M are for operators with coarity S or smaller.

This scheme could be generalised to allow the automatic disambiguation of overloaded mode names by adding submode constraints where necessary. Mode names would then refer to equivalence classes of modes under the *coincidence relation* defined as the least equivalence relation $\approx$ on modes such that:

$$\left.\begin{array}{l} M \in modes(S_1) \wedge M \in modes(S_2) \\ S \leq S_1 \wedge S \leq S_2 \end{array}\right\} \Rightarrow \quad M_{S_1} \approx M_{S_2}$$

This is based on the disambiguation scheme for function symbols suggested by Smolka (1986) (which will be discussed in Section 7.1).

Using subsorts also has implications for the unification modes of sorts. Given a sort S with subsort S' and unification mode U_S, it may be possible to unify terms in the smaller sort S' with a more efficient algorithm. To ensure that the moding of the unification predicate for sort S can still be defined with the single sort U_S, the following convention must be observed:

- If sorts S and S' have unification modes U_S and $U_{S'}$ respectively, and $S' \leq S$, then we must have $U_{S'} \leq U_S$.

However, this seems counter-intuitive. A unification algorithm for a more restricted sort should be able to work with a less restrictive mode. This problem could be eliminated by allowing a sort to have more than one unification mode. The unification modes for a sort would then be the union of the unification modes of its subsorts.

Chapter 7

The Planning System

The previous chapter identified a number of techniques that provide the facilities needed to reason in the plan specification logic presented in Chapter 5. In this chapter we describe how these techniques are combined in an implementation of a planner that is based on this logical framework.

There are two stages involved in implementing such a system. First, it is necessary to build the computational machinery needed to reason in the underlying logic of plan specifications. This then provides the functionality required to construct a tactical theorem prover that performs the proof search process outlined in Section 5.3.2. The implemented planning system reflects this two-stage structure, consisting of a preprocessor which converts an equational theory presentation defining the state specification data structure into an internal representation, and the actual planner which is a special purpose theorem prover that reasons in the resulting equational theory. The preprocessor must check a number of conditions that should hold for the theory to be meaningful and to be handled correctly during the rewriting and unification processes. These conditions are discussed below. A more detailed discussion of the form of specifications accepted by the preprocessor, together with a specification for the data type of state specifications used in the planner, is presented in Appendix B.

The system described in this chapter was developed using Edinburgh Prolog (NIP, version 1.5.02) and SICStus Prolog (version 0.6).

7.1 Defining State Specifications

State specifications are defined using order-sorted algebra with initial model semantics, as discussed in Section 6.2. The user must supply a presentation of an equational theory, consisting of a set of declarations for the sorts and function symbols of the state specification data structure together with a description of any equality relationships that hold between syntactically different pairs of terms in the resulting term algebra. An equation that holds in the theory may either be given explicitly as a (conditional) rewrite rule, or under certain conditions, it may be defined implicitly as part of a 'built-in' equational subtheory that the specification associates with a particular function symbol. By declaring each equation using one of these two techniques, the equational theory of the specification is partitioned into a set of a rewrite rules $\mathcal{R}$ and a set of equational axioms $\mathcal{E}$ which will be used during the unification and matching processes (assuming that the corresponding algorithms are built in to the system).

The declaration of 'built-in' equations is based on the function symbol *attribute* facility provided in OBJ2 and Eqlog (see Section 6.2.6). A function symbol declaration may include a list of equational attributes, each of which represents an assertion that the corresponding function obeys a particular set of axioms. For example, associating the attribute `ac` (abbreviating "associative and commutative") with a function symbol `f` is equivalent to declaring the equations `f(X,Y)=f(Y,X)` and `f(X,f(Y,Z))=f(f(X,Y),Z)`. In OBJ2, an attribute may also involve another function symbol, such as in the following declaration for an infix set union operator which is associative, commutative, idempotent and has $\emptyset$ as an identity:

$$op\ \_\cup\_\ :\ \texttt{Set}\ \texttt{->}\ \texttt{Set}\ [assoc\ comm\ idpt\ id{:}\emptyset]$$

These attributes are used by the rewrite engine which puts each term into a unique normal form that depends on the particular combination of attributes associated with each of its function symbols, and also selects an appropriate equational algorithm to use when matching each term to the left hand side of a rewrite rule.

In our system, each function symbol with equational attributes defines an equational subtheory which should be built into the system, i.e. the corresponding equational unification and matching procedures should be implemented (although these may have

moding restrictions, as discussed in Section 6.3.4). These subtheories, together with the theory consisting of the remaining function symbols and no equations, must partition the set of function symbols in the specification. Also, the equations associated with each attribute must be regular and collapse-free. These two conditions allow the (moded extension of) Yelick's unification to be used for unifying terms in the equational theory of the specification. A corresponding equational matching procedure (shown in Appendix C) is used during the rewriting process, i.e. the matching of terms to the left hand sides of rewrite rules is done modulo the built-in equational theories.

Note that this use of moded unification differs from that of Zachary: in Denali, both the modes and the equational theory are defined on a per sort basis; in our system, the modes are associated with sorts but a single sort may be partitioned into several equational subtheories through the use of these equational attributes.

The only implemented equational attributes at present are `left_commutative` (asserting that the function symbol obeys the axiom `f(A,f(B,X))=f(B,f(A,X))`), `comm` (for commutative binary function symbols) and `ac` (for associative and commutative binary function symbols, implemented for matching only and under the mode restriction that the term to be instantiated may only contain one instance of the function symbol). It is relatively easy, however, to add new unification and matching algorithms.

Function symbol attributes are also used for other purposes: the attributes `built_in` and `prolog_pred` are used to declare that a function is built-in to the system or implemented as a prolog predicate.

Operators may be overloaded, i.e. a function symbol may appear in several declarations. These may be intended to refer to a single function, as in the declarations

$$append: ne\_list \times list \rightarrow ne\_list$$
$$append: list \times ne\_list \rightarrow ne\_list$$
$$append: list \times list \rightarrow list$$

or they may define several conceptually different functions that happen to have the same name, as in

$$+ : int \times int \rightarrow int$$
$$+ : bool \times bool \rightarrow bool$$

Following the treatment in [Smolka 86], overloading is a purely syntactic concept, and the function symbols appearing in the declarations are automatically disambiguated to ensure that different functions will have different denotations in the semantics. This differs from the semantics of OBJ2 and Eqlog, where each combination of a function symbol and a rank is treated as a different operator, and the semantics forces a function symbol with 'intersecting' ranks to be well-defined on the intersection of the arities.

Smolka's disambiguation scheme uses a *coincidence relation* to partition the signature into equivalence classes of operator declarations (called *coincidence classes*). The semantics then treats each equivalence class as a separate operator denotation. Given a signature Σ, the coincidence relation is the least equivalence relation $\approx$ on Σ such that:

$$\left.\begin{array}{l} f \in \Sigma_{\bar{u},s} \ \wedge \ f \in \Sigma_{\bar{v},t} \\ \bar{w} \leq \bar{u} \ \wedge \ \bar{w} \leq \bar{v} \end{array}\right\} \ \Rightarrow \ f_{\bar{u},s} \approx f_{\bar{v},t}$$

where $\leq$ represents the extension of the partial ordering on sorts to the corresponding lexicographic ordering on finite sequences of sorts.

Operator declarations that belong to the same coincidence class must have the same attributes: if there are any differences between the sets of attributes in a coincidence class, the system will assign their union to the common operator denotation and warn the user.

A specification may include sort constraints (see Section 6.2.4.1) which are formally defined below. Retracts are not (yet) implemented, but the same effect can be achieved by explicitly adding retract operators to the signature, together with the appropriate coercion-retract cancellation equations (having the form $r_{s',s}(c_{s,s'}(X)) = X$) [Jouannaud et al. 88].

The variables appearing in sort constraints and rewrite rules do not need to be declared. Their sorts are inferred automatically by a procedure based on Smolka's sort inference function for the language TEL [Smolka 87] and the 'intended parse' function of Goguen, Jouannaud and Meseguer (1985), although it may be necessary to help this process by including extra information in term expressions (see below).

7.1.1 Restrictions on Specifications

There are a number of very natural restrictions that must be placed on equational specifications to ensure that the key results and techniques for unification and rewriting extend smoothly to the order-sorted case. The system checks that these conditions hold (except where stated otherwise) and reports if any of these tests fail. In the following discussion, the notation of Chapter 6 is used.

7.1.1.1 The Signature

The Sort Ordering

The signature must be *downward complete*. This is a restriction on the partial ordering on the sorts: any two sorts with a common subsort must have a greatest common subsort [Smolka et al. 89, §4.3].

Operator Declarations

The operators should satisfy the following *monotonicity condition*: whenever there exist declarations $f: \bar{w}_1 \rightarrow s_1$ and $f: \bar{w}_2 \rightarrow s_2$ with $\bar{w}_1 \leq \bar{w}_2$, then we must have $s_1 \leq s_2$. This condition is not needed for the underlying theoretical results to hold, but it rules out some bizarre models of the equational theory [Goguen & Meseguer 87a].

There are two other conditions that the signature must satisfy, called regularity and coregularity.

An order-sorted signature Σ is said to be *regular* if for all finite sequences of sorts $\bar{w}'$, whenever there exists a declaration $f: \bar{w} \rightarrow s$ for some (possibly overloaded) function symbol f, with $\bar{w} \geq \bar{w}'$, then there is a least rank $\langle \bar{w}_0, s_0 \rangle$ such that $f \in \Sigma_{\bar{w}_0, s_0}$ and $\bar{w}_0 \geq \bar{w}'$. This says that "any set of instances of an overloaded operator ... whose arities are bounded below ... has a member with a least arity and least sort" [Goguen & Meseguer 87a]. If this condition is satisfied then each term has a well-defined *least sort*, and it is therefore possible to disambiguate a particular application of an overloaded function symbol by determining the least sorts of its arguments, working through the term from the bottom up.

The implementation actually checks the equivalent condition [Smolka 86, Smolka

et al. 89] that there exists a partial function $lcd: \mathcal{F} \times S^* \rightarrow S$ ('least codomain') given by the following definition [Smolka 87]:

$$lcd(f, \bar{u}) = \min\{s \mid f \in \Sigma_{\bar{w},s}, \bar{w} \geq \bar{u}\}$$

For this to be well-defined, this minimum must exist whenever the set is non-empty.

A regular order-sorted signature is *coregular* if, for every function symbol f and sort s, the set $\{\bar{w} \mid f \in \Sigma_{\bar{w},t}, t \leq s\}$ is either empty or has a maximum element [Smolka et al. 89, Goguen, Jouannaud & Meseguer 85].

In general, unifying terms in order-sorted signatures has many properties in common with unsorted equational unification, and unifying two terms may require finding an infinite number of most general unifiers. The restriction to regular signatures ensures that this situation will never arise as unification in regular signatures is finitary. However, even for finite and regular signatures the problem of determining whether two terms are unifiable is an NP-complete problem [Smolka et al. 89]. By adding the conditions of downward-closure and coregularity, the situation is much improved: Meseguer, Goguen and Smolka (1989) proved that every finite, regular, coregular and downward complete signature is unitary unifying, i.e. every pair of unifiable terms has a single most-general unifier.

Sort Constraints

A signature may contain *sort constraints* to indicate that the result sorts of particular operators may be smaller than expected if certain conditions are met. These are a generalised form of the term declarations proposed by Goguen (1978), and investigated by Schmidt-Schauß (1985, 1989a) who showed that order-sorted signatures with term declarations have an undecidable unification problem. However, under the stringent conditions on sort constraints described in [Goguen, Jouannaud & Meseguer 85], the semantics of order-sorted rewriting and unification extend smoothly to this more general type of signature (although unification is still undecidable as determining the sort of a term may involve complex deductions).

Definition 7.1 Given a regular signature Σ, a *sort constraint* consists of a term $f(x_1, \ldots, x_n)$, a sort s and a set of Σ-equations C where:

1. $x_1, \ldots, x_n$ are distinct variables of sorts $s_1, \ldots, s_n$ such that there exists an operator declaration $f: \bar{w}' \rightarrow s'$ with $\bar{w}' \geq \bar{w} = s_1 \ldots s_n$ and $s' \geq s$.

2. All other declarations for f have sorts strictly bigger than s, and none has an arity smaller than $\bar{w}$.

3. No variables other than $x_1, \ldots, x_n$ appear in C.

s is called the *sort* of the constraint, and C is the *condition.*

■

A constraint is satisfied in a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$ if for all elements $a_1, \ldots, a_n$ of $\mathcal{A}$ having sorts $s_1, \ldots, s_n$ such that $C[a_1/x_1, \ldots, a_n/x_n]$ holds, the object $\mathcal{F}_f(a_1, \ldots, a_n)$ has sort s.

A signature may not contain more than one sort constraint for each operator denotation (after the disambiguation of overloaded operators).

The operational semantics outlined in [Goguen, Jouannaud & Meseguer 85] translates order-sorted specifications to equivalent specifications in a many-sorted algebra equipped with extra operators and equations (which are treated as rewrite rules). There are two types of added equations: transitivity axioms for the 'coercion' operators (which express the inclusion of sorts into their supersorts) and 'morphism rules', which describe how an operator acting on coerced arguments can be reduced to the equivalent operator restricted to a smaller domain by bringing the coercions outside the term (where they collapse into a single coercion). Given an operator $f \in \Sigma_{s_1 \ldots s_n, s} \cap \Sigma_{s'_1 \ldots s'_n, s'}$ where $s'_i \leq s_i$ for $1 \leq i \leq n$, a partition $K \cup L$ of $\{1, \ldots, n\}$, and sorts s''_l for $l \in L$ such that $s''_l \leq s'_l$, there is a morphism rule:

$$f_{s_1 \ldots s_n, s}(\ldots, c_{s'_k, s_k}(x_k), \ldots, c_{s''_l, s_l}(x_l), \ldots) = c_{s', s}(f_{s'_1 \ldots s'_n, s}(\ldots, x_k, \ldots, c_{s''_l, s'_l}(x_l), \ldots))$$

where each operator $c_{t',t}$ is a coercion from sort t' to t, all variables are different, each x_k has sort s'_k, and each x_l has sort s''_l.

When sort constraints are used, this semantics treats them as the equivalent operator declarations (temporarily ignoring the conditions), and Condition 2 above ensures that this enlarged signature remains regular. The conditions are dealt with during the transformation to many-sorted algebra by using conditional morphism rules for the operators derived from sort constraints, and by adding extra conditional equations derived from the rewrite rules (with extra conditions corresponding to the sort constraints added where necessary). The details are not given in [Goguen, Jouannaud & Meseguer 85], which states that matchings under the original and transformed theories "are equivalent for the usual cases (e.g. associativity), but the equivalence can be delicate". With these operational semantics, the evaluation of the sort constraint conditions is handled by the same mechanisms used to deal with conditional rewrite rules.

Our system does not implement the transformation to many-sorted algebra directly, as in OBJ2, but records the sorts of terms and the expected sorts of operator arguments, and this information can be used to detect when the 'sort gap' between the actual and expected sorts of a sort constraint operator requires the corresponding condition to be evaluated or added to the list of outstanding goals (this treatment of sort constraints is not yet fully implemented). The latest version of OBJ (OBJ3) uses a similar technique which is clearly described in [Jouannaud et al. 88], with the formal semantics given in [Kirchner, Kirchner & Meseguer 88]. Jouannaud et al. also show (by example) how a specification can be extended conservatively (i.e. so that equality is respected) to eliminate sort constraints by explicitly adding new operators corresponding to the sort constraint operators, specialising the equations to use these operators where necessary, and adding new conditional equations corresponding to the sort constraints. For example, eliminating the sort constraint in the stack specification fragment presented on page 111 produces the specification on the following page:

```
sorts NeStack Stack ErrStack .
subsorts NeStack < Stack < ErrStack .
op push : Nat Stack -> ErrStack .
op pushas : Nat Stack -> NeStack .
var N : Nat .
var S : Stack .
eq : push(N,S) == pushas(N,S) if length(S) < bound .
```

where `pushas` is a new operator, and the conditional equation replaces the sort constraint used previously. It is also necessary to modify the equations defining the operations `top` and `pop` (which were not included in the earlier example) to use `pushas` instead of `push`:

```
eq : top pushas(N,S) == N .
eq : pop pushas(N,S) == S .
```

This type of extension provides an alternative to implementing the machinery for handling sort constraints directly.

Modes

The extension of multi-value mode systems to order-sorted algebra that was presented in Section 6.3.5 allows a mode M declared for a sort S to be used as a mode for a subsort S' of S, and this is taken to mean the restriction of the mode M to S'. Therefore, it would cause ambiguity if a mode for a sort S were given the same name as an existing mode for any supersort of S. An error will be reported if such a declaration is made in a specification. It is, however, possible to declare a mode M for a sort S if there already exists a mode M for a *subsort* S' of S. This must be an *extension* of the mode M of S' to the larger sort S (so the condition $M_{S'} = M_S \restriction_{S'}$ should hold), although this is not checked at present.

The unification mode for a sort S must be a subsort of the unification modes for any supersorts of S. This ensures that the unification procedure for each sort is defined on a well-defined mode rather than a union of the unification modes of all its subsorts, which might not be a mode. If S does not have a unification mode declared, this constraint is

used to infer a default: the set of common submodes of the unification modes for the supersorts of S is calculated, and if there is a maximal element, this is taken to be the unification mode for S. Otherwise, an error is given unless the set is empty, in which case the default is the mode any of u (where u is the universal sort) regarded as a mode of sort S.

The system should also check that modes are closed under unification. This requires checking that the equational axioms corresponding to the declared equational attributes preserve modes, i.e. for each equation that is specified by an equational attribute associated with a particular function symbol, the terms on the left and right hand sides of the equation must belong to identical sets of modes under every possible assumption about the modes of terms that can be substituted for the variables. This test is not yet implemented.

The system does not deduce the existence of submode relationships that follow as a consequence of the declared mode ranks of operators. Therefore, the tests discussed above may incorrectly produce error messages in some cases. This can be prevented by explicitly declaring any submode relationships that exist.

The submode ordering must be downward complete. This is for implementational convenience only as it allows the same data structures to be used for sorts and modes. However, this is a very mild restriction.

7.1.1.2 Rewrite Rules

The (conditional) rewrite rules given in the specification are used to compute the values of functions on state specifications by reducing terms to their normal form. The rules should therefore be terminating and Church-Rosser modulo the equational axioms corresponding to any declared function symbol attributes (see Section 6.2.3.2). These conditions are not checked (and are undecidable in general).

The set of variables occurring in the right hand side and condition of a rule should normally be a subset of the variables in the left hand side, and a warning will be given if this condition is violated. However, there is an important class of rewrite rules where it is in fact desirable to have extra variables in the right hand side. These rewrite rules are used to transform equations to a form that can be more easily solved, and are discussed

in Section 7.2.

The rewrite rules must be *sort decreasing* [Smolka et al. 89, p. 28]: whenever $t \rightarrow_R t'$ and t has sort s, then t' must also have sort s. It is simple to test this condition as a term rewriting system is sort decreasing if and only if each of its rules is. Smolka et al. state that "in general, the key results for rewriting do not carry over to order-sorted rewriting. However, for the class of sort decreasing rewriting systems, all notations and results from unsorted rewriting generalize nicely" (their interest in this condition is for its significance in testing the confluence of unconditional rewriting systems using standard non-equational matching in finite, regular and terminating rewrite systems). OBJ2 imposes this condition to ensure that the addition of retracts preserves the Church-Rosser and termination properties of a set of rewrite rules [Goguen, Jouannaud & Meseguer 85].

The rewrite rules should also be *mode decreasing* (see Section 6.3.5.1) — this is not tested at present.

7.1.1.3 Defining Theories

The Structure of a Theory Presentation

The most basic form of theory presentation consists of declarations for:

- A set of sorts with a partial ordering defined on them. There is a predefined greatest sort `u` — the universal sort.

- A set of operators, i.e. function symbols with an associated rank. Operators may be overloaded, and may have a set of attributes associated with them, as discussed above.

- A set of modes associated with each sort, together with a partial ordering on the set of all modes such that only modes whose associated sort are related in the sort ordering can be ordered by the mode ordering.

  There is a predefined mode `any` for the universal sort `u`, which represents the set of all moded terms and is therefore a 'supermode' of any other mode for that sort (including any modes declared for the sort's subsorts). These subsort

relationships do not need to be declared. A sort S may use any mode defined for a supersort, but this must be explicitly declared as a mode for S, using the construct *mode* `of` *supersort*. There is one exception to this rule: the set of all moded terms for a sort S can be represented by using the mode '`any of u`' as a mode for the sort S, and this need not be declared. This convention eliminates the need for an implicitly defined mode `any` for each sort, as used by Zachary.

There are three reserved mode names: `atomic`, `ground` and `nonvar`. Modes with these names may be declared for any sort, and these will be interpreted directly by the system according to the usual definitions of these terms. In particular, it is not necessary to declare any mode ranks for these modes.

- A set of zero or more mode ranks for each operator. These recursively define the sets of terms that constitute each mode.
- The unification mode for each sort. This may be determined by the default rules that were discussed above.
- A set of conditional rewrite rules defining the functions corresponding to the operators in the signature.
- A set of sort constraints, subject to the conditions described above.

To help the parser disambiguate overloaded sort, mode and operator names, the user can decorate function symbols with their (numerical) arity (e.g. $f/2$), and sorts, modes, operators and terms may be qualified by the name of a theory (e.g. `sort25 of theory9`). The intended sort of a term can also be indicated to help the parser disambiguate its principal functor (e.g. `(X + Y) as int`).

Constructing Modular Specifications

Describing the state specification data structure may involve declaring many sorts, functions and equations. To ease this task, and to help the user highlight the natural structure of the specification, the system supports the development of theories in a modular fashion by providing a number of constructs for combining theories to produce more complex ones. In this section we review the specification structuring techniques

that were described in Section 6.2.5, and describe how these are incorporated into the preprocessor for our planning system.

Algebraic specifications, like computer programs, often have a natural hierarchical structure, and can best be described as a combination of a number of specifications for simpler data types, together with some additional structure. The system supports this by allowing theories to include others and combine them (without duplicating any common subparts) before enriching the result with new declarations and axioms. The included theories need not correspond exactly to existing predefined ones; it is possible to modify theories by renaming some or all of the sorts, operators and modes, and new theories can also be created by instantiating the arguments of parameterised *theory procedures* with actual theories. The use of theory procedures enables generic data structures such as lists to be defined by a single specification, thereby encouraging the maximum re-use of code. For instance, the basic functions and axioms relating to lists are independent of the type of items that may appear in the list, and therefore lists can be defined by a theory procedure which takes as a parameter a theory describing the data type of the list items.

A theory procedure is defined with respect to a number of *metatheories* — called, more intuitively, *requirement theories* in OBJ2[1] — which are a special type of theory expressing the minimum structure that must be exhibited by the actual parameters in any application of the theory procedure. These metatheories stand in for a whole class of theories that could be supplied to the theory procedure in a particular argument position. A theory procedure is applied by providing an 'actual parameter' theory to correspond to each 'formal parameter' metatheory, together with a *fitting morphism* (*view* in OBJ2) which is a structure-preserving function mapping between the metatheory and the argument theory, thus demonstrating which sorts and operators correspond to those in the metatheory.

In OBJ2 and Eqlog, the semantics of the theory structuring operations discussed above are based on constructions from category theory such as 'coproducts' and 'pushouts'. Futatsugi et al. (1984) summarise the semantics of theory procedure application as follows: "In this semantics, views correspond to theory morphisms, and

[1] As our semantics is based on that of Clear (see Section 6.2.6), we also use the associated terminology.

parameterized objects have an associated theory inclusion from their requirement theory to their body. The pushout of that theory inclusion along the view gives a new theory whose initial algebra is the desired instantiation". In category theory, a pushout is a construction for combining objects that share some common structure (the requirement theory in this case) to produce a unique 'most general' object that combines the two objects whilst identifying the common subpart. A similar technique is used to ensure that combining two theories does not result in the unnecessary duplication of common included subtheories: specifications are modelled as *based theories*, which consist of a theory sitting at the 'apex' of a set of inclusion mappings emitting from its *base* — a graph comprising particular distinguished subtheories of the apex theory, together with any inclusion relationships that exist between them. A detailed discussion of these semantics is given in [Goguen & Burstall 83], and a formal presentation of the semantics for Clear, for which these ideas were developed, is given in [Burstall & Goguen 80].

The semantics used in our system are based on the work of Sannella (1982), extended to deal with order-sorted and moded signatures. Sannella developed a more intuitive model for theories and the structuring operations offered by Clear, based on simple concepts of set theory. This enabled him to produce a more efficient (albeit less general) implementation of Clear. In Sannella's semantics, each sort and operator is 'tagged' to indicate its theory of origin. New tags are generated for the sorts and operators of a theory procedure each time it is applied, therefore preventing (for example) the sort *list* in a theory for lists of integers being confused with the sort *list* in a theory describing lists of rational numbers. The base of a theory is represented as a mapping from theory and metatheory constants to the corresponding based theories. With this approach, the combination of two theories is obtained by simply taking the union of the two signatures and bases, and the closure of the union of the two sets of equations (although in our system, we simply rely on the user composing specifications in such a way that the set of rewrite rules remains Church-Rosser and terminating). Metatheories are treated as parameterless theory procedures, and the only difference between these and ordinary theories is that a named ordinary theory includes itself in the base, whereas a metatheory does not.

There are a number of conditions that must be checked when applying the theory-

building operations described above. The fitting morphisms that are used to match the actual parameters of theory procedure applications to their associated metatheories must preserve the structure of the metatheory. This means that the sort and mode orderings, the operator attributes, and the associated equational theory must be preserved. Moreover, theory morphisms for based theories are required to preserve the base, so in a theory procedure application the bases of the actual parameters should be subsets of the corresponding metatheories — this prevents the fitting morphism from altering (e.g. by renaming sorts, etc.) any subtheories that appear in the base of a metatheory. Theory morphisms in order-sorted algebra must also preserve the ranks of operators, i.e. for each declared function symbol rank $\bar{w} \rightarrow s$, the image of the function symbol under a theory morphism σ must have a rank $\bar{w}' \rightarrow s'$ with $\bar{w}' \geq \sigma(w)$ and $s' \leq \sigma(s)$. We also require that the unification modes of sorts are preserved under theory morphisms (although this could probably be weakened to only require that the unification mode u for a sort s is mapped by a theory morphism σ to a subsort of the unification mode for $\sigma(s)$).

When the system encounters a theory procedure application in a specification, it checks all these conditions except one: it does not verify that the equational theory is preserved by the fitting morphisms. This is unsolvable in general, and it is therefore left to the user to ensure that the equations appearing in the metatheories are respected by the actual parameters.

Our renaming operation is a very restricted version of Clear's 'derive' operation which allows some of the sorts and operators of the theory to be 'forgotten' as well as possibly renaming some of those remaining. The semantics are defined using a function mapping from the newly created signature back to the original theory. The equations of the new theory are given by the inverse image under this function of the equations in the old theory. For order-sorted signatures, this would require the user to specify not only which sorts and operators should be preserved, but also which sort ordering relationships (although this problem could be alleviated by providing an alternative construct which allows the user to specify the sorts and operators to be forgotten, rather than those that should remain, as in OBJ2).

For a simple renaming operation, it is sufficient to model this as a function

transforming the signature and to create the new theory by projecting the signature and equations along this function. It is, however, necessary to remove from the base any subtheories which have sorts, operators or modes altered by the renaming function. If this is necessary, a warning is given. A warning will also be given if a renaming function is not one to one, although this is not an error provided that two distinct operator coincidence classes are not merged as a result.

At present, the renaming operation does not combine well with the use of overloaded operator symbols. All coincidence classes for a particular operator symbol and (numerical) arity combination will be renamed at once as the renaming operation is defined in terms of operator symbols, rather than coincidence classes. To correct this would require that operator expressions could be qualified by particular ranks.

7.1.2 Representing Prolog Terms and Predicates

In the semantics for plan specifications given in Chapter 5, the values appearing in the triples of a state specification were defined to be terms in the object-level language $\mathcal{L}$. This is the language used to model the objects in the world and the relationships that hold between them (via the set of Horn clauses **C**). In the planner, these value terms may appear in the literals that form part of a plan specification, and which must be satisfied during the planning process by passing them to the Prolog interpreter as goals. The values in a state specification are therefore Prolog terms, which must somehow be represented in the theory presentation that is used to define the state specification data structure. However, this theory presentation should be independent of the actual function symbols that appear in any set of clauses that is supplied to the planner, as for a given temporal language (and associated state specification data structure) the planner may be run for various assembly domains and sets of planning operators which will each have their own particular associated Prolog predicate definitions. This independence is achieved by regarding the Prolog clauses supplied to the planner as generating a built-in equational theory and by defining state specifications using a theory procedure `statespec/1` which takes such a 'prolog theory' as an argument.

The metatheory that is used as the formal parameter for the `statespec` theory procedure requires the existence of a sort `prolog_term` with modes `atomic`, `ground`,

and `nonvar`, and the boolean-valued functions `entity_type` and `type_attribute` defined for pairs of Prolog terms. The first of these corresponds to the predicate entity_type/2 mentioned in Chapter 5—the Prolog clauses must contain a fact $entity\_type(e, \tau)$ for every entity constant e, where τ is the type of the entity denoted by e. These are intended for type checking only, and therefore the (SICStus) Prolog program should include a declaration

```
:- wait entity_type/2.
```

causing calls to this predicate to be delayed until the first argument is non-variable.

The `type_attribute` function corresponds to a Prolog predicate *type_attribute/2* which is used to check that the types of entities and their attributes match (for convenience, the attributes' names are considered to be constants of the Prolog theory, rather than being conceptually distinct as defined in the semantics of Chapter 5).

In order to produce a 'compiled' form of a theory representing the state specification data structure, the `statespec` theory procedure is provided with the `prolog_theory` metatheory as a dummy argument (with the identity function as the fitting morphism).

When building the internal representation for the state specification theory procedure, the parser treats terms of sort `prolog_term` specially, letting the corresponding Prolog terms represent themselves, and isolating them from any enclosing terms by interposing a special 'wrapper' functor. This enables Prolog terms to be treated differently during unification, etc.

Besides the `entity_type` function, other boolean-valued functions operating on Prolog terms may be declared within a theory presentation and implemented as Prolog predicates. These are indicated to the system by giving them the attribute `prolog_pred`.

7.2 Solving Equations

As the logic of plan specifications is implemented using equational logic, an ability to solve equations is obviously an important requirement of the system. One possible method is to use narrowing (discussed in Section 6.2.3.2). This method is known to be complete for equational theories defined by a set of confluent and terminating rewrite rules, but is generally acknowledged to be too inefficient for practical use, except in

special cases (the investigation of narrowing techniques for various restricted classes of theories is an active research area at present). Also, for a goal-directed activity such as planning, it is undesirable for the lower levels of computational machinery to make arbitrary choices about the instantiation of variables. Such decisions should, where possible, be left to the components of the system that deal with the higher level, more strategic aspects of reasoning.

A narrowing step involves choosing a possible instantiation of a term that allows a rewrite rule to be applied. To ensure completeness, every possible choice must be explored, and even if only one solution is required, the alternatives must be remembered in case the current choice turns out to be wrong. This introduces 'don't know' nondeterminism: there may be a number of possible choices, and the system does not know which is correct. In this section we discuss the techniques the system uses to solve or evaluate equational expressions, with particular emphasis on how we attempt to avoid eager variable instantiation and the creation of choice points wherever possible.

7.2.1 Transformations on Equations

As an alternative to universal methods such as narrowing, we allow the user to define theory-specific (conditional) rewrite rules that can be used to help solve equations by rewriting them to conjunctions of simpler equations with the same set of solutions. As long as these rewrite rules preserve all solutions of an equation, and provided that the full set of rewrite rules is confluent, the application of these rules is 'don't care' nondeterministic: if more than one rule can be applied, it doesn't matter which one is chosen. However, because these rules may split an equation into a number of equations, it is necessary to relax the restriction preventing new variables from appearing in the right hand sides and conditions of rewrite rules. By introducing new variables that are shared by several of the right hand side equations, an answer substitution can be partially determined by one equation, with the unsolved part (represented by the new variables) found by solving the other equations, i.e. these variables allow the communication of partial solutions between the different equations in the right hand side. For example, in the equational theory presented in Appendix B, which represents state specifications as bags of triples constructed using a left commutative 'insert' operator ':', the following

rewrite rule is given for the equationally-defined subsort `consistent_spec` which contains all *StateSpecs* containing at most one triple for each entity–attribute pair:

$$(E, A, V_1){:}X = \mathit{update}((E, A, V_2){:}Y, Z) \;\Rightarrow$$
$$Z = (E, A, V_1){:}W \;\wedge\; X = \mathit{update}(Y, W) \;\; \textbf{if} \;\; V_1 \neq V_2$$

where the function *update* is defined so that

$$\mathit{update}(\phi, \psi) = \{(e, a, v) \in \phi \mid \nexists v'.(e, a, v') \in \psi\} \cup \psi.$$

The first equation on the right hand side states that Z must contain the triple (E, A, V_1) and binds the remaining unknown part of Z to the new variable W. The value of W is then determined by the second equation.

There are two ways of viewing rewrite rules of this type. The approach used at present is to enrich the existing equational theory with an explicit equality function symbol '=' with rank $\mathtt{u} \times \mathtt{u} \rightarrow \mathtt{boolean}$ (where u is the universal sort), and to interpret this as the identity function in the theory. These new rules can then be treated as ordinary rewrite rules, i.e. directed equations. However, we don't require that these are logical consequences of the existing equations; only that they are valid in all the intended models of the theory. In this case, we are only interested in the initial model, and therefore it is permissible to use rewrite rules that are derived using 'structural induction' on the defined data type—e.g. the rule above is only correct if every *StateSpec* is either the empty bag $\{\}$ or has the form $(E, A, V){:}B$ for some bag B. Rules that rely on the 'no confusion' property of the initial model (see page 101) may also be used; this is equivalent to making a 'closed world' assumption. For example, the following rule is used in the planner to postpone the possible introduction of 'don't know' nondeterminism by the equational unification procedure:

$$A : X = A : Y \;\Rightarrow\; X = Y$$

This rule, when regarded as a directed equation, is not satisfied by every model of the theory. For example, consider a model that maps every inconsistent *StateSpec* to some particular distinguished error value: if A, X and Y are given values such that A is inconsistent with both X and Y, then the left hand side will be satisfied by this model

while the right hand side may not be. However, in the initial model, two terms are only equal when they can be proved equal in the equational theory, and therefore the two sides are equivalent.

An alternative approach is to view these rewrite rules as inference rules that transform sets of equations to a form that can be more easily solved, and we will therefore refer to rules of this sort as *transformation rules*. This follows the treatment of unification and equation-solving techniques such as narrowing used by Martelli and Montanari (1982), Kirchner (1985), Nutt et al. (1989) and Schmidt-Schauß (1989b). In this approach, an equational problem is represented by a set of equations (or multi-equations: terms of the form $t_1 = t_2 = \cdots = t_n$) and the inference procedure is described by giving a set of transformation rules which can be applied nondeterministically to attempt to reduce the set of equations to a special *solved form* representing a correct answer substitution. For example, the following transformation can be used to express the application of a built-in unification algorithm:

$$\{u = v\} \cup \Gamma \;\Rightarrow\; \bigcup_{x \in \mathrm{Dom}(\theta)} \{x = \theta x\} \cup \Gamma$$

where θ is a most general unifier of u and v.

In order to express particular control strategies, the set of equations is sometimes divided into a solved part (corresponding to a partial answer substitution) [Martelli & Montanari 82, Hölldobler 88] or a constraint part (to which only the unification rule can be applied) [Nutt, Réty & Smolka 89], together with an unsolved part. Special *failure rules* are used to detect unsolvable sets of equations. In general, to find all solutions of the original equations, it is necessary to explore every possible sequence of rule transformations; for example, in the unification transformation above, every most general unifier of u and v must be tried in turn. However, if a rule always preserves all the solutions for a set of equations then it can be applied 'don't care' nondeterministically as a *simplification rule*. The following decomposition rule, used in the standard unification algorithm, is an example:

$$\{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup \Gamma \;\Rightarrow\; \{s_1 = t_1, \ldots, s_n = t_n\} \cup \Gamma$$

Rules such as this are said to be *complete*, but in general rules are only required to be *sound* (or *correct*). A rule $\Gamma \Rightarrow \Gamma'$ is said to be sound if every solution for Γ' is also a solution for Γ, i.e. Γ holds whenever Γ' does.

Hölldobler (1989) presents a comprehensive framework for equational logic programming using Horn clause logic with equality, based on the manipulation of sets of goal clauses using *complete sets of inference rules* in order to find a refutation proof of a query. For unification problems, this technique specialises to equation solving using *complete sets of transformations* — transformations are defined to be inference rules whose application depends on structural properties of the equation being transformed. This approach encompasses various equation-solving techniques such as paramodulation, unification and narrowing, and generalises to cover constraint logic programming, order-sorted Horn logic programming, and other related approaches. Soundness and completeness results are given for various sets of inference rules and the benefits of special types of theories (such as those defined by confluent and terminating rewriting systems) are investigated.

Compared to the uses of transformation rules described above, our equational expression rewriting rules differ in three respects. First, our rules are conditional rules that are within the theory, whereas in the standard treatment the transformation rules are unconditional meta-level rules that are generally only qualified by simple syntactic restrictions or by equality constraints (with respect to the given equational theory) that correspond to the matching implicit in our rewriting process, or to the application of 'built-in' transformation procedures such as unification. Note that this does not prevent these transformations from representing conditional narrowing or rewriting processes.

Hölldobler avoids having to treat the recursive solution of equations by considering *lazy* rewriting, narrowing and resolution rules which add the conditions as extra subgoals that are added to the transformed set of goal clauses. The eager evaluation of conditions can then be considered as a special case, where the selection function used to pick the next subgoal to be transformed always "recursively selects an equation from the conditions of the most recently used conditional rewrite rule if such conditions exist". If the conditions cannot be satisfied then execution will eventually backtrack to the point where the rule was applied. We cannot follow this approach as our transformation rules

are used as simplification rules (i.e. no backtracking information is kept), and therefore can only be applied when the condition is known to hold.

The second distinguishing feature of our transformations is that they are theory-specific (and perhaps even problem-specific: different transformation rules may be needed for different planning strategies). While the approaches described above are intended to be universal equation-solving methods — being sound and complete for any set of rewrite rules, equational programs, etc. — our transformations are defined with respect to a particular equational theory. This can be seen as following the same philosophy as Zachary's "pragmatic approach" [Zachary 88] to the problem of equational unification. As discussed in Section 6.3, Zachary's language Denali allows users to define special-purpose mode-restricted unification procedures so that the required level of complexity of the unification process can be tailored to their needs. Our equation-transforming rewrite rules have a similar role, providing the system with a shortcut for solving certain types of equational goal that the user knows will arise. This is also a natural extension of rewrite rule based languages such as OBJ2 and Eqlog, where programming takes the form of choosing an appropriate confluent and terminating set of rewrite rules, and this task will usually be guided by considerations of efficiency and suitability for the intended problem domain. It therefore seems reasonable to allow the user to also give rewrite rules that help solve equational goals concerning the defined data structures.

The third difference between our use of transformation rules and other approaches is that we are only interested in the initial model of the equational theory. Therefore, our transformations may be used as simplification rules provided that they are complete relative to the adoption of a closed world assumption and the principle of structural induction.

The symbolic equation solving program PRESS [Bundy & Welham 81, Bundy 83] used rewrite rules to solve transcendental equations such as

$$\log_e(X + 1) + \log_e(X - 1) = c$$

Isolation rules were used to decrease the depth of an unknown having a single occurrence in the left hand side of such an equation. These rules worked by applying to both sides

of the equation the inverse of the function dominating the left hand side, such as in the following rule:

$$\log_U V = W \Rightarrow V = U^W$$

Other rules were applied to terms appearing within the equation in order to prepare it for the application of isolation rules by 'attracting' and 'collecting' together different occurrences of an unknown. The control strategy used to coordinate the application of these rules in a goal-directed fashion was expressed explicitly using Prolog clauses representing meta-level axioms relating the solution of equations to the attraction, collection and isolation of terms. Searching for a solution using these meta-level rules induced a constrained object-level search through the space of possible sequences of rewrite rule applications, which was guaranteed to terminate. This technique of controlling object-level inference using meta-level rules is called *meta-level inference.* This technique could be applied to our system, but at present we rely on the programmer to ensure that the set of rewrite rules (both transformation rules and ordinary rules) is confluent and terminating, and that each transformation rule is complete, so that no backtracking information needs to be kept when a rewrite rule is applied.

7.2.2 Equational Unification

The unification procedure used in the system is based on Zachary's moded version of the algorithm for combining equational unification procedures developed by Yelick (see Section 6.2.3.1). Recall that this algorithm works by homogenising the terms being unified so that all subterms lie in the same equational subtheory as the two principal functors, performing the unification in this theory and then solving a number of unification subproblems involving the subterms that were removed during homogenisation (in practice, the initial homogenisation step can be 'folded' into the unification process). When mode restrictions are used, the algorithm is only guaranteed to be complete for pairs of moded terms with at least one belonging to the unification mode of their greatest common sort, and therefore unification will be delayed until this mode restriction is satisfied. However, it is possible that some of the subsidiary unification problems that arise will be ill-moded, and in this case Zachary's unification procedure

will exit signalling a mode failure. For our planning framework, it is better to collect these ill-moded 'unification literals' together with the substitutions corresponding to the solved part of the unification problem. These can then be added to the list of current goals when the associated partial unifier is selected.

Delaying the evaluation of unification subproblems which are not well-moded is not permitted in Zachary's framework as Denali is based on an extension of SLD-Resolution which requires that a selected literal be completely reduced before the reduction of any other literal may begin. Removing this constraint would violate the principle of predicate abstraction that Denali is designed to support — the intermediate substitutions generated may enable some implementations of a predicate to proceed while others cannot [Zachary 87, p. 39]. This is not an issue in our system.

We have extended the procedure to deal with order-sorted terms. This is straightforward provided that the signature is regular, coregular and downward complete (and therefore unitary unifying). At present, the complete set of unifiers is collected, although the algorithm can be modified to generate unifiers one at a time.

7.2.3 Applying Conditional Rewrite Rules and Sort Constraints

To apply a conditional rewrite rule $\ell \Rightarrow r$ **if** $t_1 = u_1 \wedge \ldots \wedge t_n = u_n$, the appropriate instance of the rule's condition must be a logical consequence of the given equational theory. This can be determined by normalising each t_i and u_i and testing if the resulting terms are equal with respect to the declared 'built-in' equational axioms. As discussed in Section 6.2.2, this recursive rewriting process may not terminate, but in practice the user should be able to define the rewrite rules so that this will not occur.

In Hölldobler's framework, and in some treatments of the more general technique of narrowing, the conditions are added to the proof state as additional subgoals. This approach means that the choice of which rewrite rule to apply may constrain the possible solutions, and therefore in the case of failure the system must backtrack to the point where the rule was applied. This is contrary to our principle of avoiding where possible making any decisions that would pre-empt the goal-directed reasoning performed by the tactical control level of the planner. However, this technique *is* used for ensuring that the conditions of sort constraints are satisfied when necessary. Although in

the operational semantics of order-sorted algebra (discussed in Section 7.1.1.1) sort constraints correspond to conditional rewrite rules over an extended signature, the (conceptual) application of these rules is not treated in the same way as normal rewrite rules. The system only uses sort constraints when necessary for the satisfaction of a goal, and therefore the condition must be satisfied to prevent failure. Rewrite rules, on the other hand, are applied eagerly, and therefore the conditions must be checked before rewriting can take place.

In fact, our system's treatment of rewrite rule conditions is more general than the technique described above. A rule's condition may be a conjunction $B_1 \wedge \cdots \wedge B_n$ of arbitrary boolean expressions B_i which abbreviate the equations $B_i = \texttt{true}$. If any B_i has the form t_1 = t_2, this abbreviates $(t_1 = t_2) = \texttt{true}$, where '=' is the theory's explicitly declared equality predicate (implicitly defined as if by the axiom $(X = X) = \texttt{true}$), and '=' denotes meta-level equality. Each conjunct is normalised and succeeds if reduced to `true`, otherwise if the principal functor of the resulting term is a built-in predicate (declared via the `built_in` attribute) the corresponding predicate is called to reduce the term. At present, the only built-in predicates are '=' and '\=': a term t_1 = t_2 reduces to `true` if t_1 and t_2 are equal in the defined equational theory, and otherwise remains unreduced (in which case the rule cannot be applied), whereas t_1 \= t_2 reduces to `false` if t_1 and t_2 are equal, to `true` if they are non-unifiable, and is otherwise not reduced.

The same approach is used to solve or decompose goals in the proof state which have built-in predicates as their principal functors, but the corresponding (Prolog) predicates may be different for this mode of use. For example, currently an '=' goal results in the unification procedure being called, whereas a '\=' goal is reduced to `false` if the two terms are not unifiable, to `true` if the terms belong to the empty equational theory so that a SICStus dif/2 constraint can be soundly asserted, and to `false` (on backtracking) if this constraint fails. If none of these apply, then the goal remains unreduced. The restriction on the use of a dif/2 constraint is necessary as otherwise (for example) a goal $\{x, y\}$ \= $\{y, x\}$ would succeed.

This treatment of built-in predicates in conditions and goals offers the possibility of allowing the user to have more control over the situations in which variables may be

instantiated. This is only permitted when solving goals at present, but by introducing separate operator symbols to correspond to these different modes of use (e.g. having both '==' and '=', where one allows instantiation and the other does not) it would be possible for the user to specify when instantiation could be permitted during the evaluation of conditions.

Another possible extension would be to allow the user to declare that certain rewrite rules could be used for narrowing, or to allow rewrite rules to introduce special 'don't care' variables into the reduced terms, and to allow these to be instantiated during the 'matching' phase of future rewrite rule applications.

7.3 The Theorem Prover

Before the planner is run, the theory presentations discussed in Section 7.1 are processed to generate an internal data structure. The state specification theory is, in general, dependent on the particular temporal operators that are to be used. This is because adding inference rules for new temporal operators may require the formal specification of any new functions that appear in the conclusions of the rules. Therefore, for each temporal language used, the appropriate 'compiled' form of the state specification theory must be loaded before the planner is run, and this contains the information required to parse the goal plan specification, to determine the modes and associated unification theories of terms to be unified, and to rewrite terms to their canonical form after each proof step. The associated inference rules must also be preprocessed and loaded into the planner when required.

As discussed in Section 5.3.2, planning in this framework involves composing a proof state with an inference rule to produce a new proof state. The planner's proof state consists of the original goal plan specification, the current subgoal plan specifications, the Prolog goals that have not yet been satisfied, and a list of equality constraints.

The operational semantics of rewriting is based on that used in OBJ2, as discussed above (see page 136): the order-sorted logic is represented in unsorted logic by adding extra operators that convey the sorts of terms (using a technique that is essentially equivalent to OBJ2's use of coercion functions), but the 'collapsing' of

nested coercions, etc. is handled directly by the unification and matching procedures, rather than by the addition of extra rewrite rules.

As each theory may include several instances of the rewrite rules declared for a particular theory (e.g. by including several applications of a theory procedure), each rewrite rule is stored in the Prolog database in a generic form, indexed by a unique key. The internal model of the state specification theory represents each instance of a rewrite rule by the appropriate key and the required bindings for the sorts and operator denotations (i.e. coincidence classes). Following the technique used in OBJ2 and Eqlog, all rewrite rule instances which have the same operator outermost on the left hand side are linked together in a list. The rewrite engine will apply each of these rules as long as it is successful, and will then cycle through the list trying each rule in turn until the principal function symbol of the term has changed.

At present, the unification procedure returns complete sets of unifiers, allowing higher levels of control to choose a preferred unifier. An alternative approach would be for the unification procedure to return a stream of unifiers, with the calling procedure discarding unifiers until a satisfactory one is found. However, this could result in the proof search strategy being incomplete (although generally, completeness may not be obtainable in planning anyway). Another possibility is to provide the unification procedure with a partial substitution expressing the desired constraints on any unifier.

No explicit support for a tactical level of reasoning is currently provided. The search for a proof can be controlled by writing tactics or other forms of control strategy as Prolog predicates that directly examine the proof state and inference rules and call the predicates that interface with the unification and rewriting procedures.

Chapter 8

Planning in the Logical Framework

The previous chapter described how the techniques of Chapter 6 were combined to produce an implementation of a specialised theorem prover for planning using the logical framework developed in Chapter 5. This chapter demonstrates how the system is used to describe and solve a simple planning problem, and discusses how existing planning techniques can be modelled in the framework. The discussion in this chapter is based on the equational theory for state specifications presented in Appendix B.

8.1 Describing the Problem

The first step required in any planning methodology is to construct an appropriate formal model of the problem domain. For our system, this involves choosing a suitable model of the world in terms of entities, attributes and values, defining the relevant behavioural operators using plan specification axioms, and providing a Prolog program containing the required entity_type/2 facts (which declare the types of the entity constants) and the definitions for any Prolog predicates that appear in the plan specification axioms. For example, in the Soma assembly domain, the Prolog program would include predicates to generate and test possible configurations for a part to fit into an assemblage, tests for part stability, and predicates to calculate and transform robot grasps. In the simple example discussed below, no such predicates are needed.

Before planning commences, it is also necessary to decide which temporal operators may be used to construct plans and to derive the corresponding inference rules (if

not done previously). The state specification theory presentation may then need to be updated with definitions for any new functions and rewrite rules that are required.

Once all this is done, the state specification theory presentation and the inference rules can be parsed and preprocessed and loaded into the planning component of the system. The goal plan specification is then entered into the system. This is the theorem to be proved and has the form $\langle \phi_i, P, \phi_f \rangle$, where ϕ_i and ϕ_f are state specifications describing the initial and goal states, and P is a plan term which will normally be a variable or a partially instantiated term containing plan variables. As the semantics of plan specifications are defined by an implication which is vacuously true whenever either of the two state specifications is inconsistent, the two *StateSpecs* are parsed as the sort consistent_spec (see Appendix B) to ensure that only meaningful solutions are found. This is a subsort of the sort statespec and is defined using a sort constraint asserting that only state specifications having at most one entry for any entity–attribute pair are members of this sort. This has the effect of adding the extra conditions consistent(ϕ_i) and consistent(ϕ_f) as goals in the initial proof state. For the same reason, a conjunction of disequality constraints (denoted by $distinct(|\phi_i| \cup |\phi_f|)$ in Chapter 5) is automatically added to the initial proof state to ensure that all distinct entity symbols appearing as the first element of some triple in ϕ_i or ϕ_f denote different entities. This goal (eventually) reduces to a conjunction of constraints of the form $entity\_sym1 \neq entity\_sym2$, which the system reduces to true after asserting SICStus dif/2 constraints between the two arguments if they are unifiable (the operator '$\neq$' is denoted '\=' in the equational theory given in Appendix B). These constraints are treated as assumptions: they may enable the application of rewrite rules that are conditional on the inequality of terms (determined by testing for non-unifiability) and if any such constraints are still blocked at the end of planning, they are simply ignored.

For some planning strategies it may be desirable to leave one (or both) of the state specifications in the goal *PlanSpec* as variables, with equational logic constraints (discussed in the next section) given as additional goals to restrict their possible values. For instance, in the example below, when a backwards planning strategy is used the initial state specification variable ϕ_i is constrained by a goal of the form $\phi_i \subseteq \psi$, where ψ is a state specification describing the initial state of the planning problem.

The initial *PlanSpec* formula to be proved may not contain any Prolog literals, i.e. it must have the form $\langle \phi, P, \psi \rangle$. Instead, the user may provide a set of Prolog goals to be evaluated during the course of planning. These allow Prolog predicates to be called to instantiate variables appearing as values in the initial plan specification. This is useful if a state specification must have complex terms appearing in it, such as in the Soma world where the goal plan specification must describe the initial configurations of the parts and the shape of the assemblage. These goals would normally be intended to be called before planning commences, but could alternatively be kept for evaluation during the planning process along with any other Prolog goals that may result from the use of the atomic actions' *PlanSpec* axioms. Logically, these are treated as conjuncts in the formula to be proved, and so the initial proof state has the form[1]:

$$\frac{\Delta_1 \quad \cdots \quad \Delta_m \quad \Gamma \quad \Gamma \triangleright \langle \phi_i, P, \phi_f \rangle \quad C_1 \quad \cdots \quad C_n}{\Delta_1 \wedge \cdots \wedge \Delta_m \wedge \langle \phi_i, P, \phi_f \rangle \wedge C_1 \wedge \cdots \wedge C_n}$$

where $\Delta_1, \ldots, \Delta_n$ are the user-supplied Prolog goals, $C_1, \ldots C_n$ are the initial equational constraints, and Γ is a variable that will become instantiated to a conjunction of Prolog goals during the course of the planning (these will be the goals associated with the various *PlanSpec* axiom instances that are used).

8.2 The Planning Process

At any stage of planning the planner maintains a list of current subgoals, the solution of which will guarantee the validity of (an instantiated form of) the original goal plan specification. These subgoals come in three different types: plan specification formulae, Prolog goals, and equations that must be satisfied. The first two types are formulae of the planning logic developed in Chapter 5; the third type of goals are part of the equational logic that is used to implement state specifications, and in the following discussion they will be treated as equational 'constraints' that are conceptually distinct from the subgoals of the current 'proof state' (as defined in Chapter 5). Note that a constraint consisting of a boolean-valued term such as $X \subseteq Y$ is shorthand for the equation $(X \subseteq Y) = \text{true}$.

[1] This corresponds to the initial proof state presented on page 94 for the general framework (without equational constraints) developed in Section 5.3.2.

Planning proceeds as discussed in Section 5.3.2: by decomposing plan specification subgoals using the temporal operator 'introduction' inference rules in reverse, and by evaluating Prolog goals when suggested by the tactical reasoner that determines the planning strategy. To illustrate how the equational logic implementation of state specifications supports this process, we consider a simple planning problem—the block swapping problem presented in Section 4.2.1 (page 66)—and show how it can be solved using the forwards and backwards planning strategies that were pictured schematically in Figure 5.2.

In the discussion below, a slightly abbreviated version of the state specification syntax of Appendix B is used: state specifications are constructed from the empty bag $\{\}$ and entity–attribute–value triples of the form (E, A, V), using the left-commutative 'insert' operator ':'. Variables are denoted by Greek and upper case Roman letters (e.g. ϕ, A), and bold variable symbols ($\boldsymbol{\phi}$, **A**, etc.) are used to abbreviate particular predefined terms or the values bound to variables that have been instantiated (e.g. the symbol $\boldsymbol{\phi}$ represents the binding of the formerly uninstantiated variable ϕ).

Recall that for the block swapping problem, the world is modelled as consisting of objects of the two types block and table. Objects of these types have the attributes loc (giving the location of the block), and on (specifying the object that is on the table; nil if there is none) respectively. The only temporal operator available is the sequencing construct ';'.

The initial plan specification describing the problem has the form $\langle \phi_i, P, \phi_f \rangle$ where

$$\begin{aligned} \phi_i &\equiv (A, \mathrm{loc}, T_1){:}(B, \mathrm{loc}, T_2){:}(T_1, \mathrm{on}, A){:}(T_2, \mathrm{on}, B){:}(T_3, \mathrm{on}, \mathrm{nil}){:}\{\} \\ \phi_f &\equiv (A, \mathrm{loc}, T_2){:}(B, \mathrm{loc}, T_1){:}(T_1, \mathrm{on}, B){:}(T_2, \mathrm{on}, A){:}(T_3, \mathrm{on}, \mathrm{nil}){:}U \end{aligned}$$

and A and B are entity variables of type block and T_1, T_2 and T_3 are of type table (in Appendix B, entity symbols are represented by terms of the form entity_sym$type, but for the sake of brevity, we omit the type information here). The variable U in the 'tail' of ϕ_f represents any 'side effects' of the generated plan, i.e. effects that were not explicit goals of the problem. This is required due to the semantics of plan specifications which state that the postcondition *StateSpec* describes *all* the changes to

the initial state caused by the plan. However, for many problems (including this one), there will be no extra entity–attribute pairs that could have values in the postconditions, and so this variable could be safely replaced by $\{\}$.

Parsing these state specifications as the sort consistent_spec will require using the sort constraints for ':' and '$', and this will result in the goals consistent(ϕ_i), consistent(ϕ_f) and entity_type(A, block), etc. being added to the initial proof state. Also, as discussed above, a conjunction of disequality constraints (abbreviated by the meta-level expression $distinct(\phi_i \cup \phi_f)$) is also added to assert that each of A, B, T_1, T_2 and T_3 must represent a distinct entity (of course, a goal such as $A \neq T_1$ will succeed immediately as these entity variables have different types). These added goals act as constraints on the possible instantiations of variables, and can help to cause early failure on some branches of the search tree.

There is only one planning operator in this problem domain: the generic action move(X, T, T'), which has preconditions described by the state specification

$$(X, \text{loc}, T):(T, \text{on}, X):(T', \text{on}, \text{nil}):\{\}$$

and postconditions given by

$$(X, \text{loc}, T'):(T, \text{on}, \text{nil}):(T', \text{on}, X):\{\}$$

A forwards planning strategy: One simple planning strategy is to plan forwards from the initial state by decomposing the initial plan variable P into the sequence $a\,;P'$ where a is some action that can be performed in ϕ_i (either move(A, T_1, T_3) or move(B, T_2, T_3) in this case). This choice can be guided by giving preference to any actions with postconditions that intersect the state specification describing the desired goal state (although for the initial step in this example, there are no such actions). The selection of an appropriate action is the responsibility of the tactical level of the planner, but even without this guidance, the process of unifying state specifications will help to constrain the possible operators that can be applied, and to instantiate the parameters of generic actions. For this example, choosing the generic action move(X, T, T') results

in the proof tree

$$\frac{\langle \boldsymbol{\alpha}_1, \text{move}(X,T,T'), \boldsymbol{\beta}_1 \rangle \qquad \langle \gamma_1, P', \delta_1 \rangle}{\langle \boldsymbol{\phi}_i, \text{move}(X,T,T') ; P', \boldsymbol{\phi}_f \rangle}$$

together with the equations

$$\boldsymbol{\phi}_i = \boldsymbol{\alpha}_1 \cup (\gamma_1 \setminus \boldsymbol{\beta}_1) \tag{1}$$

$$\boldsymbol{\phi}_f = \text{update}(\boldsymbol{\beta}_1, \delta_1) \tag{2}$$

where

$$\boldsymbol{\alpha}_1 \equiv (X, \text{loc}, T){:}(T, \text{on}, X){:}(T', \text{on}, \text{nil}){:}\{\}$$

$$\boldsymbol{\beta}_1 \equiv (X, \text{loc}, T'){:}(T, \text{on}, \text{nil}){:}(T', \text{on}, X){:}\{\}$$

A semantic version of the ';'-introduction rule's syntactic side condition (see Section 5.2) is also added to the proof state. This is the constraint:

$$\text{consistent}(\boldsymbol{\beta}_1 \cup \gamma_1)$$

Although it will be incrementally reduced and satisfied during planning (causing the assertion of various SICStus dif/2 constraints), this constraint will play no role in the solution of this particular planning problem and will therefore be ignored in the following discussion.

After several applications of the rewrite rule $A{:}X \cup Y \Rightarrow A{:}(X \cup Y)$ to the right hand side of (1) above, this equation reduces to

$$(A, \text{loc}, T_1){:}(B, \text{loc}, T_2){:}(T_1, \text{on}, A){:}(T_2, \text{on}, B){:}(T_3, \text{on}, \text{nil}){:}\{\}$$
$$= (X, \text{loc}, T){:}(T, \text{on}, X){:}(T', \text{on}, \text{nil}){:}\,(\gamma_1 \setminus (X, \text{loc}, T'){:}(T, \text{on}, \text{nil}){:}(T', \text{on}, X){:}\{\})$$

(replacing $\boldsymbol{\phi}_i$ and $\boldsymbol{\alpha}_1$ with their definitions). This is a well-moded unification problem with two (partial) solutions, the first of which consists of the substitution $\{T_1/T,\ T_3/T',\ A/X\}$ together with the ill-moded unification subproblem

$$(B, \text{loc}, T_2){:}(T_2, \text{on}, B){:}\{\} = \gamma_1 \setminus (A, \text{loc}, T_3){:}(T_1, \text{on}, \text{nil}){:}(T_3, \text{on}, A){:}\{\} \tag{1'}$$

The second solution is similar, with A and B, and T_1 and T_2 interchanged.

Several applications of the rewrite rules

$$A:X = Y \setminus B:Z \;\Rightarrow\; A:W = Y \setminus Z \;\wedge\; X = W \setminus B:\{\} \qquad (*)$$

and $(X \setminus A:Y) \setminus Z \;\Rightarrow\; (X \setminus Y) \setminus A:Z$ reduce (1′) to give a conjunction of the two equations

$$\gamma_1 = (B, \text{loc}, T_2):(T_2, \text{on}, B):W \qquad (1a)$$

$$\{\} = W \setminus (T_3, \text{on}, A):(T_1, \text{on}, \text{nil}):(A, \text{loc}, T_3):\{\} \qquad (1b)$$

where W is a new variable. Note that the rewrite rule $(*)$ is not valid for bags in general, as these can contain duplicate entries (although if the condition $A \neq B$ were added to the rule, it would be). Therefore, in Appendix B this rule is restricted to apply to consistent *StateSpec* terms only.

Equation (1a) is 'solved' by instantiating γ_1, and the rule $X \setminus Y = \{\} \;\Rightarrow\; X \subseteq Y$ rewrites (1b) to give the constraint

$$W \subseteq (T_3, \text{on}, A):(T_1, \text{on}, \text{nil}):(A, \text{loc}, T_3):\{\} \qquad (1b')$$

(this is really just 'syntactic sugar', as the rewrite rules for '$\subseteq$', notated as '=<' in Appendix B, are mostly direct counterparts of similar rules for '$\setminus$').

Now, three applications of the conditional rewrite rule

$$(E, A, V_1) : X = \text{update}((E, A, V_2) : Y,\ Z)$$
$$\Rightarrow\; Z = (E, A, V_1) : W \;\wedge\; X = \text{update}(Y, W) \;\text{ if }\; V_1 \neq V_2$$

reduce Equation (2) to the conjunction

$$\delta_1 = (A, \text{loc}, T_2) : W_1 \;\wedge\; W_1 = (T_1, \text{on}, B) : W_2 \;\wedge\; W_2 = (T_3, \text{on}, \text{nil}) : W_3$$
$$\wedge\; (B, \text{loc}, T_1):(T_2, \text{on}, A):U = \text{update}(\{\}, W_3)$$

The term $\text{update}(\{\}, W_3)$ reduces to W_3, and satisfying each conjunct by binding W_1, W_2, W_3 and δ_1, we get

$$\boldsymbol{\delta}_1 \equiv (A, \text{loc}, T_2):(T_1, \text{on}, B):(T_3, \text{on}, \text{nil}):(B, \text{loc}, T_1):(T_2, \text{on}, A):U$$

The first planning step is now complete: the initial decomposition of the goal plan P into the compound plan $\text{move}(X, T, T')\,;P'$ has resulted in X being unified with

A, T with T_1, and T' with T_3, and all terms in the current proof state as well as the outstanding constraint (1b′) are normalised. There is one possible backtracking point corresponding to the alternative bindings for X, T, and T'. The tactical level of the planner must now step in to suggest a decomposition for the subgoal $\langle \gamma_1, P', \delta_1 \rangle$, by choosing an appropriate next action that is performable in a state satisfying γ_1, using the 'goal' state specification δ_1 to guide this selection. Although γ_1 is only partly instantiated by Equation (1a), the possible values that its 'tail' W can take are greatly restricted by the sub-bag constraint (1b′). To make use of this information, the tactics must explicitly look for constraints of this form (although the system could facilitate this by keeping them in a separate list or by associating such '$\subseteq$' constraints with the constrained variable).

In fact, in forwards planning it would do no harm to convert the '$\subseteq$' to an '$=$' by binding W to the 'bounding' *StateSpec* on the right hand side of (1b′) — this is the only constraint on W, and it conveys the information that although the previous action has 'achieved' the triples (T_3, on, A), $(T_1, \text{on}, \text{nil})$ and (A, loc, T_3), these effects may not in fact be necessary for any future actions to succeed. As these triples are known to hold anyway, (possibly) strengthening the precondition γ_1 in this way will have no adverse effect on the later stages of planning. For the same reason, it does not matter if the initial precondition *StateSpec* ϕ_i is stronger than is absolutely necessary for a plan to be found; it simply means that the resulting plan specification formula (but not the plan itself) will be less general than it could have been.

Comparing δ_1 and γ_1, it is now clear that an appropriate next action is $\text{move}(B, T_2, T_1)$ (having shown in the previous step how the parameters of generic actions can be instantiated through unification, to shorten this discussion we now assume that the planner's tactical level will make a suitable choice). The subplan P' is therefore decomposed into the sequence $\text{move}(B, T_2, T_1)\,;P''$ by unifying the plan specification subgoal $\langle \gamma_1, P', \delta_1 \rangle$ with the conclusion

$$\langle \alpha_2 \cup \gamma_2 \setminus \beta_2,\ \text{move}(B, T_2, T_1)\,;P'',\ \text{update}(\beta_2, \delta_2) \rangle$$

of the corresponding instance of the ';'-introduction rule, where α_2 and β_2 are the pre- and postcondition state specifications for $\text{move}(B, T_2, T_1)$. This requires solving the two equations:

$$\gamma_1 = (B, \text{loc}, T_2):(T_2, \text{on}, B):(T_1, \text{on}, \text{nil}):\{\}$$
$$\cup\ \gamma_2 \setminus (B, \text{loc}, T_1):(T_2, \text{on}, \text{nil}):(T_1, \text{on}, B):\{\} \quad (3)$$

$$(A, \text{loc}, T_2):(T_1, \text{on}, B):(T_3, \text{on}, \text{nil}):(B, \text{loc}, T_1):(T_2, \text{on}, A):U$$
$$= \text{update}((B, \text{loc}, T_1):(T_2, \text{on}, \text{nil}):(T_1, \text{on}, B):\{\}, \delta_2) \quad (4)$$

subject to the additional constraint $\text{consistent}(\beta_2 \cup \gamma_2)$ (which plays no part in the solution of this planning problem).

Expanding γ_1 in Equation (3), normalising the right hand side using the rule $A:X \cup Y \Rightarrow A:(X \cup Y)$, and applying the rule $A:X = A:Y \Rightarrow X = Y$ to the resulting equation (the rewrite rules for '=' are applied before unification is attempted), we get

$$W = (T_1, \text{on}, \text{nil}):(\gamma_2 \setminus (B, \text{loc}, T_1):(T_2, \text{on}, \text{nil}):(T_1, \text{on}, B):\{\})$$

With this binding of W, the rule $A:X \subseteq A:Y \Rightarrow X \subseteq Y$ rewrites the constraint (1b′) to give

$$\gamma_2 \setminus (B, \text{loc}, T_1):(T_2, \text{on}, \text{nil}):(T_1, \text{on}, B):\{\} \subseteq (T_3, \text{on}, A):(A, \text{loc}, T_3):\{\}$$

which further reduces to

$$\gamma_2 \subseteq (T_1, \text{on}, B):(T_2, \text{on}, \text{nil}):(B, \text{loc}, T_1):(T_3, \text{on}, A):(A, \text{loc}, T_3):\{\}$$

under several applications of the rule

$$X \setminus (E, A, V):Y \subseteq Z \Rightarrow X \setminus Y \subseteq (E, A, V):Z \ \textbf{if}\ \text{no\_entry}(E, A, Z)$$

where the condition $\text{no\_entry}(E, A, Z)$ is defined to be true if and only if the *StateSpec* Z contains no triples with entity E and attribute A. This last rule is restricted to consistent state specifications, although it is valid for bags in general even without the condition. This ensures that the *StateSpec* $(E, A, V):Z$ constructed on the right hand side of the rule is consistent.

From Equation (4), the three rules

$$(E, A, V) : X = \text{update}(Y, Z)$$
$$\Rightarrow \; Z = (E, A, V) : W \;\wedge\; X = \text{update}(Y, W) \;\; \textbf{if} \;\; \text{no\_entry}(E, A, Y),$$

$$(E, A, V_1) : X = \text{update}((E, A, V_2) : Y, Z)$$
$$\Rightarrow \; Z = (E, A, V_1) : W \;\wedge\; X = \text{update}(Y, W) \;\; \textbf{if} \;\; V_1 \neq V_2$$

and $X = \text{update}(X, Y) \Rightarrow Y \subseteq X$

give us

$$\delta_2 \equiv (A, \text{loc}, T_2) : (T_3, \text{on}, \text{nil}) : (T_2, \text{on}, A) : V$$

and

$$V \subseteq (T_1, \text{on}, B) : (B, \text{loc}, T_1) : \{\}.$$

At this stage, comparing δ_2 and the constraint on γ_2, it is clear that the action $\text{move}(A, T_3, T_2)$ can complete the plan to give

$$P = \text{move}(A, T_1, T_3) \,;\, \text{move}(B, T_2, T_1) \,;\, \text{move}(A, T_3, T_2)$$

A backwards planning strategy: An alternative simple planning strategy is to first select an action that will achieve one or more of the goals of the final state specification (represented as triples), determine what preconditions must hold for the remaining triples to also hold after this action is performed, and use the resulting state specification to represent the goal state in a subsidiary planning problem.

For this strategy (and using the equational theory for state specifications presented in Appendix B), the most general form of the initial plan specification is $\langle \phi_i, P, \phi_f \rangle$ where

$$\phi_f \equiv (A, \text{loc}, T_2) : (B, \text{loc}, T_1) : (T_1, \text{on}, B) : (T_2, \text{on}, A) : (T_3, \text{on}, \text{nil}) : U$$

and ϕ_i is a variable constrained by the goal

$$\phi_i \subseteq (A, \text{loc}, T_1) : (B, \text{loc}, T_2) : (T_1, \text{on}, A) : (T_2, \text{on}, B) : (T_3, \text{on}, \text{nil}) : \{\}$$

(as noted before, the precondition state specification in a *PlanSpec* does not describe the initial state: it need only include those triples that are *necessary* for the plan to

proceed).

Let $\mathbf{S_i}$ denote the term on the right hand side of this constraint.

Inspection of ϕ_f shows that only two actions can be performed to reach this state: move(A, T_3, T_2) and move(B, T_3, T_1). Choosing the first of these, and decomposing P as the sequence P'; move(A, T_3, T_2) leads to the two equational problems

$$\phi_i = \alpha_1 \cup (\gamma_1 \setminus \beta_1) \tag{1}$$

$$\phi_f = \text{update}(\beta_1, \delta_1) \tag{2}$$

where

$$\gamma_1 \equiv (A, \text{loc}, T_3){:}(T_3, \text{on}, A){:}(T_2, \text{on}, \text{nil}){:}\{\}$$

$$\delta_1 \equiv (A, \text{loc}, T_2){:}(T_3, \text{on}, \text{nil}){:}(T_2, \text{on}, A){:}\{\}$$

Instantiating ϕ_i according to (1) makes the sub-bag constraint above become

$$\alpha_1 \cup (\gamma_1 \setminus \beta_1) \subseteq \mathbf{S_i}$$

which reduces to a conjunction of the constraints

$$\alpha_1 \subseteq \mathbf{S_i} \qquad \text{and} \qquad \gamma_1 \setminus \beta_1 \subseteq \mathbf{S_i}.$$

Equation (2) reduces under the rules

$$\text{update}(X, A{:}Y) \Rightarrow A{:}\text{update}(X \setminus\setminus A{:}\{\}, Y),$$

$$(E_1, A_1, V_1){:}X = Y \setminus\setminus (E2, A_2, V_2){:}Z$$
$$\Rightarrow (E_1, A_1, V_1){:}W = Y \setminus\setminus Z \ \wedge \ X = W \setminus\setminus (E_2, A_2, V_2){:}\{\},$$

and $$(X \setminus\setminus A{:}Y) \setminus\setminus \Rightarrow (X \setminus\setminus Y) \setminus\setminus A{:}Z$$

(where the operator '$\setminus\setminus$', defined by $X \setminus\setminus Y = \{(e, a, v) \in X \mid \nexists v'.(e, a, v') \in Y\}$, removes from X any triples whose entity and attribute have a value assigned in Y) to give the two equations:

$$\beta_1 = (B, \text{loc}, T_1){:}(T_1, \text{on}, B){:}W$$

$$U = W \setminus\setminus (T_2, \text{on}, A){:}(T_3, \text{on}, \text{nil}){:}(A, \text{loc}, T_2){:}\{\}$$

The first of these is solved by instantiating β_1, and from above we have $\gamma_1 \setminus \beta_1 \subseteq \mathbf{S_i}$ which reduces to a conjunction of the equation

$$W = (A, \text{loc}, T_3){:}(T_3, \text{on}, A){:}(T_2, \text{on}, \text{nil}){:}W'$$

(where W' is a new variable) with a constraint of the form $\{\} \subseteq \_$ (which reduces to true) under the rewrite rules

$$X \setminus A{:}Y \Rightarrow X \setminus Y \quad \textbf{if} \quad \text{not\_in}(A, X)$$

and

$$A{:}X \subseteq Y \Rightarrow Y = A{:}Z \wedge X \subseteq Z$$

We now have $\alpha_1 \subseteq \mathbf{S_i}$ and

$$\beta_1 \equiv (B, \text{loc}, T_1){:}(T_1, \text{on}, B){:}(A, \text{loc}, T_3){:}(T_3, \text{on}, A){:}(T_2, \text{on}, \text{nil}){:}W'$$

which suggests a possible decomposition of P' into $P''\,;\text{move}(B, T_2, T_1)$. The computations of the next step progress in a similar fashion to the last one, giving $\alpha_2 \subseteq \mathbf{S_i}$ and

$$\beta_2 = (A, \text{loc}, T_3){:}(T_3, \text{on}, A){:}(T_1, \text{on}, \text{nil}){:}V$$

where V is unknown. From this, it is easily seen that the problem can be solved by choosing $\text{move}(A, T_1, T_3)$ as the initial action.

The final proof tree resulting from this example is shown in Figure 5.1 (page 91).

■

Note that the forward and backward planning strategies discussed above are not the only ones possible for linear planning. In fact they are both special cases of the more general strategy of decomposing a plan P into a sequence $P'\,;a\,;P''$ (this corresponds to the 'means-ends' strategy used by GPS [Newell & Simon 63] and STRIPS [Fikes & Nilsson 71]). An inference rule can be derived for this by composing two applications of the ';'-introduction rule to give:

$$\frac{\langle \alpha, P', \beta\rangle \qquad \langle \phi, P, \psi\rangle \qquad \langle \gamma, P'', \delta\rangle}{\langle \alpha \cup (\phi \setminus \beta) \cup (\gamma \setminus (\beta \cup \psi)), P'\,;a\,;P'', \mathit{update}(\beta, \mathit{update}(\psi, \delta))\rangle}$$

Using this inference rule would require the introduction of a 'do nothing' action

(no_op say), interpreted as the identity function on states, which could be used to instantiate P' (or P'') if a should turn out to be the first (or last) action of the plan.

8.3 Extensions of the Framework

8.3.1 Extending Plan Specifications

For some temporal representations it may be necessary to represent more information in state specifications. For example, it is not possible to give a compositional inference rule corresponding to the concurrent plan composition operator $\|$ without finding some way of expressing the constraint that the plans being composed may not interfere with each other. A compositional $\|$-introduction rule, with conclusion $\langle \phi, P \| Q, \psi \rangle$ say, must (by definition) treat the subplans P and Q as if they were atomic actions, ignoring their inner structure. For atomic P and Q, the semantics (which model concurrency by interleaving) give the interpretation of $P \| Q$ as:

$$I(P \| Q) = \mathcal{I}_{\|}(\{P\}, \{Q\}) = \{PQ, QP\}$$

Now, suppose that we instantiate P to $a\,;b$ and Q to $c\,;d$. We then get $I((a\,;b) \,\|\, (c\,;d)) = \{abcd, cdba\}$ whereas the semantics give

$$I((a\,;b) \,\|\, (c\,;d)) = \mathcal{I}_{\|}(\{ab\}, \{cd\}) = \{abcd, acbd, acdb, cabd, cadb, cdab\}$$

It seems that any inference rule for '$\|$' must take the inner structure of the subplans into account! However, if P and Q are constrained to be non-interfering, then any interleaved execution path of $P \| Q$ has exactly the same effect as a non-interleaved one, and so a compositional inference rule will be sound — in the example above, any state specification that holds for the sequences $abcd$ and $cdba$ will also hold for all sequences in $\{abcd, acbd, acdb, cabd, cadb, cdab\}$. Unfortunately, it is not sufficient to test the subplans' pre- and postconditions to determine non-interference: consider the two plans

$$\text{move}(A, T_1, T_2) \qquad \text{and} \qquad \text{move}(B, T_3, T_2)\,;\,\text{move}(B, T_2, T_3)$$

There is no conflict between the pre- or postconditions of these plans, but there is a clear possibility of a collision at table T_2.

A simple and practical way to solve this problem is to introduce some notion of the 'region of influence' of a plan, and to explicitly represent this in plan specifications. A possible basis for this is given in [Lucassen 87], which describes a framework for integrating imperative and functional programming language constructs in a single language while preserving the benefits of both approaches. This is precisely what is required for reasoning about concurrent actions in planning problems: a plan is like an imperative program, altering the environment as a result of its actions, whereas reasoning about concurrent actions, program transformation, etc. is greatly facilitated in a functional language.

In Lucassen's framework, a region is a set of memory locations and every expression has an associated *effect* specification describing in abstract terms (*allocate*, *read* or *write*) the ways in which it interacts with any declared regions. Inference rules are given to describe how the effects (and types) of compound expressions can be inferred from those of their components. Program fragments can declare and use *private* regions to hide the side effects of any internal data manipulations from the outer parts of the program. To ensure that concurrent program execution remains correct, *monitored* regions may be declared, and program expressions may be specified as *critical sections* for a given monitored region.

A similar scheme for assembly robotics could help to reduce the complexity of planning as it supports the hiding of inessential information. For example, if the robot requires somewhere to temporarily place a part, the planner does not need to know where this will be — it is only necessary that a suitable place can be found at the time, and that this will be free of interference from other agents and actions while it is being used. In fact, taken to its limits, this approach could allow programs to be viewed as functional programs that would be amenable to such techniques as data-flow analysis (where the entities in the work-cell are the data) and efficient concurrent execution. For our present purposes, however, we simply note that including a representation of abstract regions in plan specifications would allow plans that affect disjoint regions to be treated as atomic actions when reasoning about their concurrent execution. The flexible

nature of our planning framework with its declarative definition of data structures such as state specifications, and its operations defined in terms of transformations on proof states (see Section 5.3.2), should make this type of extension straightforward.

8.3.2 Extending the Plan Representation Scheme

There are some types of temporal representations commonly used in planning systems which cannot be expressed using compositional operators that take plan terms as arguments. For example, non-linear planners such as TWEAK [Chapman 87] work with a partial order, incrementally adding and modifying the ordering constraints between actions as planning progresses. Some other planning formalisms (e.g. [Drummond 85]) can represent conditional plans and sensing actions. To cater for these more expressive temporal representations, the plan specification logic can be extended to allow temporal operators to take arbitrary first order (i.e. Prolog) terms as additional arguments, and to allow predicates involving these to appear as premises in inference rules. Thus, a non-linear plan could be expressed by a plan term

$$\mathrm{do}([P_1, \ldots, P_n], PO)$$

where $[P_1, \ldots, P_n]$ is a list of plan terms and PO is a Prolog data structure representing a partial order over the set of integers $\{1, \ldots, n\}$. This would represent a nondeterministic plan, interpreted in the semantics by the set of all action sequences satisfying the ordering constraints in PO.

In general, nonlinear planners produce plans that are intended for sequential execution; the partial ordering is used to reduce the amount of search the planner must do by allowing actions to be represented as unordered relative to each other. Chapman (1987) has given a precise formal statement of the conditions under which this type of plan is correct with respect to a given initial state and set of goals. His *modal truth criterion* (MTC) is a modal logic formula defining when a goal is necessarily true after the execution of a non-linear plan, but it can also be interpreted as a nondeterministic procedure for adding constraints to a partial plan to ensure that it achieves a given goal, and this was the basis for Chapman's planner TWEAK. By repeatedly applying this

procedure until all goals are satisfied, new actions and ordering constraints are added, and equality or disequality constraints involving actions or objects are asserted, until a correct plan is produced or the process fails and backtracks.

In our framework there are several possible ways in which the MTC could be used to implement non-linear planning. The neatest would perhaps be to work directly at the level of proof state transformations (see Section 5.3.2 where the application of inference rules was described in these terms), providing a rule corresponding to each of Chapman's plan modification operators. Alternatively, it may be possible to express these transformations as inference rules, with Prolog predicates appearing in the premises to manipulate the partial order data structure.

Conditional plans could be dealt with by introducing an operator test with two arguments: a Prolog predicate representing the condition to be tested, and a term representing the result of the test (either true or false when ground). This would be interpreted as the identity function on all states satisfying the test, and as a special error function for states in which the test fails (mapping every state to an unsatisfiable error state — this requires a slight extension to the semantics). The following inference rule could then be used for conditional plans:

$$\frac{\langle \phi,\ \mathrm{test}(C, \mathrm{true})\,;P\,, \psi\rangle \qquad \langle \phi,\ \mathrm{test}(C, \mathrm{false})\,;Q\,, \psi\rangle}{\langle \phi,\ \mathrm{if}(\mathrm{test}(C, \_), P, Q)\,, \psi\rangle}$$

If a number of different temporal operators are used, there may no longer be a unique form for a plan, due to redundancy in the combined temporal language. This could be dealt with by introducing a set of proof state transformations designed to reduce every plan term to a normal form. For instance, the no_op operator introduced at the end of the previous section could be eliminated from a plan term by applying a proof state transformation that replaces every plan term of the form no_op ; P or P ; no_op with P. Another possible transformation would perform the following replacement on plan terms:

$$d\,;\mathrm{do}([a, b, c], PO_1) \ \Rightarrow\ \mathrm{do}([a, b, c, d], PO_2) \ \textbf{if}\ \mathrm{add\_initial\_action}(PO_1, PO_2)$$

where the condition would be evaluated as a Prolog goal to create the new partial order PO_2.

Chapter 9

Conclusion

9.1 Summary

In this thesis we have proposed a framework for robotic assembly planning, designed to suit the requirements of a behaviour-based assembly control system. A simple but structured world model is used, consisting of a number of entities of different types, their attributes of interest, and the (possibly complex) values associated with these. This enables the inherent structure of the problem domain to be modelled in an abstract form while hiding as much detail from the planner as possible — this is part of the philosophy of the behaviour-based approach, and the SOMASS system has demonstrated how effective this can be. However, robotic assembly problems may involve reasoning about complicated spatial and geometrical relationships, and to support this, our framework splits the specifications of operators and plans into two parts: the representations of the states of the world before and after the corresponding action is performed, together with a set of first order literals which are evaluated as Prolog goals in order to test preconditions and generate terms to appear in the world model. Thus, complex computations can be performed without introducing the inefficiencies of general purpose theorem proving.

The logical form of our plan specification formulae is novel, with the descriptions of actions and plans represented by a single type of formula. This is like a Hoare logic, but with semantics that incorporate a form of frame axiom similar to the STRIPS rule. This, together with the simple and structured world model enables (equational)

unification to be used to propagate the information in world states.

Another issue considered in this thesis is the representation of time for the robotic assembly domain. Many different temporal representation schemes have been proposed for planning and other problems involving time and processes. However, most systems are built around a particular model of time and cannot be easily extended to cope with new types of temporal constraint. Robotic assembly has its own particular features that influence the choice of temporal representation, and various researchers have pointed out different aspects, e.g. Fox and Kempf (1985a) have commented on the benefits of having a lot of redundancy in the possible ordering of actions at run time, while Lyons (1986) states that the domain of robotics is fundamentally concurrent in nature. Although there are some temporal representations that are sufficiently powerful to represent any sort of temporal relationships that may arise (e.g. full first order temporal logics), the complexity of reasoning with them rules out most practical applications, although the work of Lansky (see page 18) may provide a partial answer to this problem. Instead of taking this approach, our framework is designed to allow users to choose their own temporal operators and to support the development of planning strategies for them. However, these temporal operators must be able to be described in terms of compositional inference rules. While this may seem a harsh restriction, it is part of the general philosophy of our approach that the problem must be structured as much as possible to avoid the inherent intractability of planning. With this viewpoint, this seems to be a natural restriction, and it encourages (and in fact forces!) the user to exploit the inherent structure of the domain to make the planning problem amenable to formal analysis, at least for restricted cases.

In order to support reasoning with different temporal languages, the planning framework is based around the technique of tactical reasoning, commonly used for interactive theorem proving. This provides a flexible basis for experimenting with new representations and planning strategies.

Of course, there is no point in having a flexible logical system if it cannot be implemented efficiently. Therefore, a large part of this thesis has been devoted to discussing techniques to implement the required computations on state specifications. The techniques of order-sorted equational specification, rewriting and moded unification

9.2 Relation to Other Work

This section briefly surveys a number of existing planning techniques and systems and how they relate to this work. In particular, we concentrate on a discussion of the frame problem and how our solution is related to previous techniques. The description of the various techniques discussed below is based on the presentation in [Georgeff 87].

The first planning system was built by Green (1969), who investigated the application of the (then) fledging technique of resolution theorem proving to problems involving changes of state. His logical model of state and action was based on McCarthy's predicate calculus formulation [McCarthy & Hayes 69], which in its later incarnations has become known as the *situation calculus*. In this formalism, the effects of actions are described using axioms such as the following for a block stacking action $puton$:

$$holds(clear(A), S) \wedge holds(clear(B), S) \wedge A \neq B$$
$$\Rightarrow holds(on(A, B), result(puton(A, B), S))$$

where *holds* denotes a satisfaction relation between propositions and states, and *result* is a function taking an action and a state, and producing the state resulting from performing that action in the state.

This axiom describes the preconditions and effects of the action, but it does not completely define the resulting state. For example, the above axiom does not specify the *colour* of the blocks A and B after the $puton$ action. To express the fact that the colour of a block remains the same after stacking another block on top, an explicit *frame axiom* must be given:

$$holds(colour(B, red), S) \Rightarrow holds(colour(B, red), result(puton(A, B), S))$$

In the situation calculus, frame axioms such as this are needed for every action

and for all properties that are unaffected by that action. The difficulties resulting from having to specify and reason with this large number of axioms are collectively known as the *frame problem.* Planners based on uniform proof procedures, such as Green's system, have been found to be hopelessly inefficient for most problems due to the increased size of the search caused by frame axioms. Although there are techniques that can help to reduce this problem (see, e.g. [Genesereth & Nilsson 87]), most researchers have concentrated on trying to find formalisms that don't require frame axioms.

The planner STRIPS [Fikes & Nilsson 71] introduced a different representation of operators. Instead of representing the effects of actions as conditions that are guaranteed to hold after the action is executed, STRIPS describes actions as operators that perform simple (syntactic) modifications to the data base of facts representing the current state of the world. Each operator is associated with three lists of formulae: a precondition list, an *add* list and a *delete* list. The corresponding action can be performed if the precondition formula is true in the current state, and the representation of the resulting world state is obtained from the old one by adding the formulae in the add list (which are known to be true as a result of the action) and deleting the formulae in the delete list (which may no longer be true in the new state). Lifschitz (1987) showed that this computation is only sound if a particular set of *allowable* formulae is selected, and the descriptions of world states and the add and delete lists are restricted to only contain formulae from this set.

If we consider the case where the allowable formula may have one of the two forms $a(e) = v$ and $e_1 \neq e_2$, it can be seen that (for linear plans) our plan specification formulae are a special case of STRIPS operator declarations, where the preconditions are split into a set of allowable formulae and a set of arbitrary literals. For a *PlanSpec* in maximal form, this correspondence would have the delete list equal to the allowable formula part of the preconditions (i.e. the formulae in the precondition *StateSpec* are deleted) and the add list would correspond to the postcondition *StateSpec.*

The backward planning strategy demonstrated in our planning examples (and illustrated in Figure 5.2a). corresponds to the technique of *goal regression.* Waldinger (1977) extended Warren's (1974) form of regression to allow a goal G to be passed backwards through an action A appearing in a plan in order to generate a new goal,

which if satisfied in A's initial state will guarantee that G holds after A is performed. Rosenschein investigated regression and its forward counterpart *progression*, using a dynamic logic (see page 27). He pointed out that regressed (and progressed) goals correspond to the notion of weakest provable preconditions (strongest provable postconditions) that are used in the verification of programs using program logics — such as Hoare logic, to which our plan specification logic bears a strong resemblance.

In general, it is not always possible to compute regressions and progressions of arbitrary goals. Rosenschein's planner has a general procedure for computing regressions, which Chapman (1987) has described as "unworkably inefficient". With our logic, these are computed as part of the equational unification process. Genesereth and Nilssons' (1987) discussion of regression in the STRIPS framework contains an equation on sets of goals that is equivalent to the equation $\phi = \alpha \cup (\gamma \setminus \beta)$ that arises during the application of the ';'-introduction rule in our system. This seems to be for descriptive purposes only, rather than for reasoning with as in our system.

9.3 Further Work

In this section a number of ideas for further investigation and development of the planning framework are suggested. These are split into three areas: planning techniques, theoretical development and implementation techniques.

Planning Techniques

This planning framework was designed to provide a general tool for investigating a number of aspects of robotic assembly planning; in particular, the framework should support the use of various temporal representation schemes and the development of associated planning strategies for them, and should also provide an appropriate model for studying the implications of the behaviour-based assembly paradigm on the planning process. These issues remain largely unexplored at present.

At present, the main scope for further work is to gain experience with the use of other temporal operators and the implementation of different planning strategies using tactics. A number of extensions to the framework were proposed in Section 8.3 to allow

temporal operators with non-compositional semantics to be expressed in the framework, and this would be an interesting area for future development.

A useful test-bed for these ideas could be provided by developing a Soma-world planner in this framework. The on-going development of the SOMASS assembly system will provide a rich domain for investigating many aspects of planning. Also, attempting to model the existing SOMASS planner would raise many issues such as how hierarchical planning could be best expressed in this framework, and how problem-specific optimisations — such as using failure-directed backtracking between the Prolog goals — could be implemented.

In Chapter 4 a notion of transformations with respect to a particular assembly strategy was introduced to explain the relationship between the levels of the SOMASS planner. Also, the question was raised as to whether a general assembly planner in this framework could be transformed into a task-specific optimised form. These issues should be investigated further.

Theoretical Aspects

There are several aspects of the framework that could benefit from further theoretical development or a more unified treatment. In particular, the use of unification and equation-transforming rewrite rules would benefit from being recast as inference rules in Hölldobler's framework for equational logic programming (see Section 7.2.1). This would give a principled basis for incorporating any other equation-solving techniques such as narrowing.

The integration of Zachary's multi-valued mode system needs some more consideration. If new functions are added to the state specification theory presentation it may become necessary to introduce separate declarations for the mode restrictions on equational matching algorithms. At present, the unification and matching modes are assumed to be the same, although this is not the case for left-commutative function symbols, for which the matching algorithm is restricted (for ease of implementation and efficiency) to only deal correctly with 'pattern' terms containing at most one occurrence of the function symbol. This mode restriction is not checked at present, but this is not necessary as none of the current rewrite rules' left hand sides break this restriction.

It may also be necessary to allow a unification procedure to have more than one unification mode. Whereas Zachary's sorts correspond to his equational subtheories, ours do not as there may be multiple function symbols with declared equational attributes for a single sort. As mode signatures define modes on a sort-by-sort basis, it may be difficult to sum up the mode restrictions for all equational subtheories within a sort by specifying a single unification mode.

Implementation techniques

It is important for this type of planning system that the low-level computations on state specifications, such as the rewriting of terms, be as efficient as possible. At present, this is rather slow. Futatsugi et al. (1984) describe a number of techniques used in the rewrite engine of OBJ2 that could be applied to our system. However, for our system the main cause of inefficiency is the need to apply the rewrite rules to all terms in the proof state after each inference step in case they are no longer in normal form as a result of some variable becoming instantiated. It would be much better if rewrite rules could be triggered by the instantiation of a variable within a term. One possible approach to be investigated is to use freezing of rewriting goals together with destructive assignment to cause the replacement of a term with an equationally equal term whenever instantiation causes an appropriate rewrite rule to activate. There are complications with this approach, however, due to the possibility of overlapping rewrites being applicable.

Appendix A

Proof of the Sequence Operator Inference Rule

In this appendix we prove the soundness of the inference rule for the sequence operator ';' which was defined in section 5.2:

$$\frac{\Gamma \triangleright \langle \alpha, P, \beta \rangle \qquad \Delta \triangleright \langle \gamma, Q, \delta \rangle}{\Gamma, \Delta \triangleright \left\langle \alpha \cup (\gamma \setminus \beta),\ P;Q,\ update(\beta, \delta) \right\rangle} \quad \text{if } \beta \cup \gamma \text{ is a consistent } \textit{StateSpec}$$

We show this by demonstrating that an interpretation and a valuation that together satisfy the two antecedent plan specifications also satisfy the conclusion. Note that we are only considering *consistent* plan specifications.

Theorem Let $\mathfrak{M} = \langle \mathcal{M}, \mathcal{F} \rangle$ and $\mathcal{V}$ be an interpretation and a valuation that satisfy the consistent plan specification formulae $\Gamma \triangleright \langle \alpha, P, \beta \rangle$ and $\Delta \triangleright \langle \gamma, Q, \delta \rangle$.

Then if $\beta \cup \gamma$ is a consistent state specification, the plan specification formula $\Gamma, \Delta \triangleright \langle \alpha \cup (\gamma \setminus \beta),\ P;Q,\ update(\beta, \delta) \rangle$ is satisfied by $\mathfrak{M}$ and $\mathcal{V}$. Also, if $\alpha \cup (\gamma \setminus \beta)$ is consistent, and the antecedent *PlanSpecs* are in maximal form, then the conclusion plan specification is consistent and in maximal form.

Proof Let model $\mathfrak{M} = \langle \mathcal{M}, \mathcal{F} \rangle$ and valuation $\mathcal{V}$ satisfy $\Gamma \triangleright \langle \alpha, P, \beta \rangle$ and $\Delta \triangleright \langle \gamma, Q, \delta \rangle$, and suppose that the following conditions hold:

$$\models_{\mathcal{M}} distinct(|\alpha \cup (\gamma \setminus \beta)| \cup |update(\beta, \delta)|)\,[\mathcal{V}] \tag{1}$$

$$\text{and} \quad \models_{\mathcal{M}} \Gamma, \Delta\,[\mathcal{V}] \tag{2}$$

Now, $|\alpha| \subseteq |\alpha \cup (\gamma \setminus \beta)|$ and from the definition of *update* it follows that $|\beta|, |\delta| \subseteq |update(\beta, \delta)|$.

Also,

$$\begin{aligned} |\gamma| \subseteq |(\gamma \setminus \beta) \cup \beta| = |\gamma \setminus \beta| \cup |\beta| &\subseteq |\gamma \setminus \beta| \cup |update(\beta, \delta)| \\ &\subseteq |\alpha \cup (\gamma \setminus \beta)| \cup |update(\beta, \delta)| \end{aligned}$$

Therefore, $(|\alpha| \cup |\beta|) \cup (|\gamma| \cup |\delta|) = |\alpha \cup (\gamma \setminus \beta)| \cup |update(\beta, \delta)|$ and so we must have $\models_{\mathcal{M}} distinct(|\alpha| \cup |\beta|)\,[\mathcal{V}]$ and $\models_{\mathcal{M}} distinct(|\gamma| \cup |\delta|)\,[\mathcal{V}]$ or (1) would not hold.

From (2) we have $\models_{\mathcal{M}} \Gamma\,[\mathcal{V}]$ and $\models_{\mathcal{M}} \Delta\,[\mathcal{V}]$, and applying Definition 5.2 for the two antecedent *PlanSpecs* we get:

$$\forall \sigma \in \Sigma \;\; \forall \bar{p} \in I(P) \quad Sat_{\mathcal{M},\mathcal{V}}(\sigma, \alpha) \Rightarrow f_{\bar{p}}(\sigma) = update\_state(\sigma, \beta) \tag{3}$$

$$\forall \sigma \in \Sigma \;\; \forall \bar{q} \in I(Q) \quad Sat_{\mathcal{M},\mathcal{V}}(\sigma, \gamma) \Rightarrow f_{\bar{q}}(\sigma) = update\_state(\sigma, \delta) \tag{4}$$

Now, let $\sigma \in \Sigma$ and $\bar{r} \in I(P;Q) = \mathcal{I}_{(;)}(I(P); I(Q))$. Then, from Definition 5.1, $\bar{r} = \bar{p}\,\bar{q}$ where $\bar{p} \in I(P)$ and $\bar{q} \in I(Q)$.

Suppose $Sat_{\mathcal{M},\mathcal{V}}(\sigma, \alpha \cup (\gamma \setminus \beta))$. Then we have $Sat_{\mathcal{M},\mathcal{V}}(\sigma, \alpha)$ and from (3) it follows that $f_{\bar{p}}(\sigma) = update\_state(\sigma, \beta)$. Also, $Sat_{\mathcal{M},\mathcal{V}}(\sigma, \gamma \setminus \beta)$ and as $\beta \cup \gamma$ is consistent, $Sat_{\mathcal{M},\mathcal{V}}(update\_state(\sigma, \beta), \gamma \setminus \beta)$ and therefore $Sat_{\mathcal{M},\mathcal{V}}(update\_state(\sigma, \beta), \gamma)$ must hold. From (4) we get

$$f_{\bar{q}}(update\_state(\sigma, \beta)) = update\_state(update\_state(\sigma, \beta), \delta)$$

But $f_{\bar{q}}(update\_state(\sigma, \beta)) = f_{\bar{q}}(f_{\bar{p}}(\sigma)) = (f_{\bar{p}} \circ f_{\bar{q}})(\sigma) = f_{\bar{r}}(\sigma)$ and it remains to show that $update\_state(update\_state(\sigma, \beta), \delta) = update\_state(\sigma, update(\beta, \delta))$, which is left as an exercise for the reader.

Finally, we note that the conclusion is a meaningful plan specification if $\alpha \cup (\gamma \setminus \beta)$

is consistent, as the consistency of $update(\beta, \delta)$ follows easily from its definition and the consistency of β and δ. Also, if the postcondition state specifications in the premises are given in their maximal form, so that for every entity e and attribute a appearing in a triple in α (respectively γ) there is a triple (e, a, v') in β (δ), then the conclusion plan specification also has this form. This follows easily by case analysis.

■

Appendix B

A Theory Presentation for the State Specification Data Structure

This appendix presents an algebraic specification for the data structure of state specifications that is used in the planning system described in Chapter 7. It is a direct description of the conceptual structure of state specifications as defined in Chapter 5 and is therefore less efficient (but more perspicuous) than other implementations that have been considered. For example, an alternative representation could be based upon modelling state specifications as mappings from entities to other mappings that associate values with attributes, and by extending a technique presented by Zachary (1987) it should be possible to automatically translate between terms of the 'denotation theory' presented here and the equivalent terms in a more efficient 'representation theory'.

The equational theory presented below models state specifications as *bags* (or multisets) of triples. This representation is equivalent to sets for *consistent* state specifications, provided that for each such *StateSpec* ϕ, we are only interested in interpretations of the function and variable symbols in ϕ that map syntactically different entity symbols to distinct elements in the interpretation's domain. To see this, consider a representation of state specifications using terms of the form $(e_1, a_1, v_1){:}\ldots{:}(e_n, a_n, v_n){:}\{\}$ where ':' is a bag 'insert' operator satisfying the law of left-commutativity ($A : B : X = B : A : X$) and '$\{\}$' is a constant representing the empty bag (this is an abbreviated form of the representation used in the equational theory presentation below). If the extra axiom $A : A : X = A : X$ were added,

this would be a representation for sets; however under the restrictions outlined above, this extra axiom is not required, as no consistent *StateSpec* term appearing in a plan specification can be instantiated by the system to produce a ground bag containing more than one copy of any element.

Suppose that a term ρ representing a consistent state specification contains the triples (e_i, a_i, v_i) and (e_j, a_j, v_j). The syntactic consistency restriction requires that $e_i \not\equiv e_j$ or $a_i \not\equiv a_j$ (where '$\equiv$' denotes syntactic identity). If $e_i \equiv e_j$ then we must have $a_i \not\equiv a_j$, and as state specifications are defined to have constant attribute symbols only, no substitution can make these terms equal in the initial model of the theory. If $e_i \not\equiv e_j$, then $\models_{\mathcal{M}} e_i \neq e_j\,[\mathcal{V}]$ for any interpretation $\mathcal{M}$ and valuation $\mathcal{V}$ assigning different values to e_i and e_j. It therefore follows that if each pair of syntactically distinct entity symbols in ρ are constrained to be unequal (i.e. if $distinct(|\rho|)$ holds), then (e_i, a_i, v_i) and (e_j, a_j, v_j) can never become equal through instantiation and so ρ can be represented by a bag. Now, every entity symbol appearing in ρ will also appear in the *PlanSpec* being proved ($\langle \phi, P, \psi \rangle$ say) and so $|\rho| \subseteq |\phi| \cup |\psi|$. As discussed in Section 5.2, the conjunction of disequality constraints denoted by $distinct(|\phi| \cup |\psi|)$ can be added as an assumption in the initial proof state without invalidating the proof—it only prevents valid but meaningless *PlanSpecs* from being generated— and therefore, provided this is done, bags can be safely used to represent state specifications.

In the equational theory presented below, the boolean-valued function `consistent` implements the semantic consistency constraint.

To facilitate the definition of world models for new problem domains, attribute names are considered to be Prolog constants. These should be declared for each type using Prolog `type_attribute/2` facts (this treatment differs from the semantics given in Chapter 5 where the attribute names were considered to be conceptually distinct from the first order language $\mathcal{L}$).

B.1 The Syntax of Equational Presentations

The preprocessing component of the planning system is responsible for parsing specifications and building an internal representation of the resulting equational theory. At

present, each of the various declarations composing a specification is presented as a Prolog fact (or unit clause), although ideally a system such as this would read and parse specifications directly from a file or the terminal. Before presenting the specification, it is necessary to explain the notation and conventions used in these declarations.

A specification consists of presentations for a number of named theories of three different types: ordinary theories, theory procedures and metatheories (discussed in Chapter 7). These theories may be defined recursively by fitting together existing theories using a number of theory-building operations, and possibly enriching the result with new sorts, operators, modes, sort and mode ordering constraints, and equations. The operations provided for constructing new theories from existing ones are theory combination, procedure application (where the formal parameters of a theory procedure are 'instantiated' with actual theories that 'fit' the associated metatheories), and signature changing (by renaming the sorts, operators and modes). The (unnamed) theories that can be built using these operations are represented by *theory expressions*, which can have one of the following forms:

theory_constant
theory_exp + *theory_exp*
theory_proc_name(*theory_proc_args*)
theory_exp * *renaming_morph*
identifier = *theory_exp*

where

- '+' represents theory combination.
- *theory_proc_args* is a sequence of one or more terms of the form *theory_exp* / *fitting_morph*, separated by commas.
- *fitting_morph* is either a constant that has been declared as a fitting morphism or a list of terms of the form *sig_element_type*(*elt_exp* `is` *elt_exp*) where *sig_element_type* is `sort`, `op` or `mode`, and *elt_exp* is an expression denoting a sort (syntactic category *sort_exp*), operator symbol (*op_exp*), or a mode (*mode_exp*) and has one of the following forms: *name*, *name* `of` *theory_identifier*, or

`u_mode(`*sort_exp*`)` (used to refer to the unification mode of a sort without explicitly naming it). *theory_identifier* is a theory constant or the keyword `enrichment` — this allows the user to refer specifically to a function symbol/rank declaration in the current theory enrichment (i.e. the set of new declarations).

- *renaming_morph* has the same form as a fitting morphism, except '`is`' is replaced by '`to`' (this is intended to highlight the different uses of these two types of morphism).

- An '`=`' expression has the value of the theory expression that appears on the right, and as a side effect, the identifier on the left is declared as a new theory constant having this value, with its scope limited to the current theory presentation.

Fitting morphisms are declared by a clause of the form:

`fitting_morph(`*identifier*`,` *fitting_morph*`).`

The explicit use of fitting morphisms could be largely eliminated by adopting the conventions for default and abbreviated 'views' used in OBJ2 and Eqlog [Goguen 84]. The simplest such convention is to allow the omission of entries for sorts, modes or operators whose images under the morphism have the same name, but at present this scheme is not implemented. However, it is not necessary to specify the images for the sorts, modes and operators that lie in the theory's base, as these must be unchanged under any theory morphism. Also, the unification mode for each sort must be unchanged under the mapping, so these entries in the mapping are assumed by default.

Each theory being defined must be declared by a clause specifying its name and a list of *included theories* which are combined to form the nucleus of the new theory. The declarations for the three types of theories have the following forms:

`theory(`*theory_constant*`,` *included_theories*`).`
`theory(`*theory_proc_name*`(`*formal_params*`),` *included_theories*`).`
`meta_theory(`*theory_constant*`,` *included_theories*`).`

where

- *included_theories* is a list of expressions of the form `using(`*theory_exp*`)`, `protecting(`*theory_exp*`)`, or `extending(`*theory_exp*`)`. These correspond to

the three ways of importing modules in OBJ2 (see page 115), although the conditions associated with `protecting` and `extending` are not checked — these assertions simply provide a mechanism for the user to declare the intended mode of use for imported theories. Every ordinary theory (except `boolean`) and theory procedure automatically includes the theory `boolean`.

- *formal_params* is a comma-separated sequence of terms of the form *variable* : *theory_constant* such that *theory_constant* denotes a metatheory.

The variables in the formal parameters of a theory procedure declaration may appear as actual parameters for theory procedure applications occurring in the included theory list — these variables are treated as the corresponding metatheories when 'compiling' the theory procedure, and so this type of partially uninstantiated theory procedure application must be supplied with fitting morphisms that map between metatheories: from the included theory procedure's metatheories to the metatheories of the associated formal parameter variables. For example, if `triv` is a metatheory that is used to express the requirement that a theory has at least one sort, and we wish to create a theory procedure for generic sets, building upon a theory procedure for bags (declared as `bag(_:triv)`), then the following theory declaration is required:

```
theory(set(T:triv), [using(bag(T / triv_id))]).
```

where `triv_id` is a predeclared fitting morphism which is the identity function on `triv`. When the theory procedure `set/1` is applied, `T` becomes instantiated to an actual theory (or, more accurately, the corresponding compiled theory procedure is extended along the supplied fitting morphism) to produce the appropriate specialised theory of sets.

For each declared theory, the desired enrichment of the combined included theories is specified by a number of facts declaring the new sorts, operators, modes, equations and sort and mode ordering constraints. These declarations are associated with a particular theory, theory procedure or metatheory by giving the generic form of the corresponding theory expression as one argument of the fact. This *theory schema* is either a theory constant (in the case of an ordinary theory or a metatheory) or a

term *theory_proc_name*$(X_1, \ldots, X_n)$ where each of the X_i is a distinct variable. The possible types of declaration are:

`new_sort(`*theory_schema*`,` *modes*`,` *sort_name*`).`

modes is a list of atoms naming the modes to be associated with the new sort. The universal sort `u` is visible in all theories and does not need to be declared.

`new_modes(`*theory_schema*`,` *sort_exp*`,` *modes*`).`

modes is a list declaring new modes for the (existing) sort denoted by *sort_exp*. There is a predefined mode called `any` for the universal sort `u`. This represents the set of all moded terms and may be used as a mode for any sort without declaring it. The system also has built-in procedures for testing for membership of the built-in modes `atomic`, `ground` and `nonvar`. These modes must be explicitly declared when required for a sort. This is necessary because the mode `any` of `u` contains all other modes, and the user may want (for example) the ground terms of some sort to remain unmoded. However, no mode ranks need to be declared for these modes.

`subsorts(`*theory_schema*`,` *subsort_dec_forest*`).`

subsort_dec_forest is the empty list or a list of the form `[`*sort_exps* `<<` *subsort_dec_forest* `|` *subsort_dec_forest*`]` where *sort_exps* is a sort expression or a list of sort expressions (N.B. the Prolog operator `<<` has been changed from its standard definition to become right associative — the declarations for this and the other Prolog operators used appear at the end of this appendix). All sorts are automatically less than the universal sort `u`.

`submodes(`*sort_exp*`,` *theory_schema*`,` *submode_decs*`).`

submode_decs has the same form as the subsort declarations (but with mode rather than sort expressions). All modes are predeclared to be less than the mode `any` of `u`.

`unification_mode(`*sort_exp*`,` *theory_schema*`,` *mode*`).`

mode must be one of the declared modes for the given sort. A default mode is calculated (see Chapter 7) if there is no unification mode declared for a sort.

`function(`*op_exp*`,` *theory_schema*`,` *rank*`,` *attributes*`).`

This declares a (new) rank for a (possibly pre-existing) function symbol, with the attributes specified in the list *attributes*. *rank* is an expression of the form '*sort_exp* `x` ··· `x` *sort_exp* `--->` *sort_exp*', or '`--->` *sort_exp*' (for constants).

`mode_rank(`*op_exp*`,` *theory_schema*`,` *rank*`,` *mode_rank*`).`

This type of declaration specifies a mode rank for the operator denoted by *op_exp*. The rank is included to help determine the appropriate instance of the operator. *mode_rank* has a similar form to a rank expression, but involves mode rather than sort expressions.

`rewrite_rule(`*theory_schema*`,` *op_exp*`,` *rule*`,` *cond*`).`

This declares a conditional rewrite rule, with the rule body *rule* (having the form *term_exp* `==>` *term_exp*) and the condition *cond* (which is a boolean-valued term or a conjunction of such terms using the Prolog operator ','). *op_exp* together with the (numerical) arity of the term on the rule's left hand side should uniquely determine the correct operator denotation to use for parsing this term. The syntactic category *term_exp* represents a term that may be optionally qualified by a theory name (using '`of`') or by a sort name (using '`as`') to help disambiguate the principal functor.

`sort_constraint(`*theory_schema*`,` *op_exp*`,` *rank*`,` *term_exp*`,` *cond*`).`

This declares a sort constraint with the specified operator, rank and condition. This says that any instance of the term denoted by *term_exp* has the sort specified in *rank* when the boolean expression *cond* holds.

Finally, we present the specification for the state specification data structure, followed by the Prolog operator definitions used. The `statespec` theory is intended to be loaded into the planning component of the system using the theory expression:

```
statespec(prolog_theory)
```

B.2 The Theory Presentation

```
%%%%%%%%%%%%%%%%%%%
%%% Theory univ %%%
%%%%%%%%%%%%%%%%%%%

theory(univ, []).
% has 'universal' sort u by default

fitting_morph(univ_as_triv, [sort(elt is u),
                             mode(u_mode(elt) is any of u)]).

%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Metatheory triv  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%

meta_theory(triv, []).

new_sort(triv, elt, [u_mode]).

unification_mode(elt, triv, u_mode).


%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory boolean  %%%
%%%%%%%%%%%%%%%%%%%%%%%%

% always available to other theories

theory(boolean, []).
new_sort(boolean, boolean, [ground]).
unification_mode(boolean, boolean, ground).

function(true, boolean, ---> boolean, []).
function(false, boolean, ---> boolean, []).

function( =, boolean, u x u ---> boolean, [comm, built_in]).
function(\=, boolean, u x u ---> boolean, [comm, built_in]).
function(&, boolean, boolean x boolean ---> boolean, [ac]).

rewrite_rule(boolean, &, true & X ==> X, true).
rewrite_rule(boolean, &, false & _ ==> false, true).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Metatheory prolog_theory   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

meta_theory(prolog_theory, [protecting(boolean)]).

new_sort(prolog_theory, prolog_term, [nonvar, atomic, ground]).
submodes(prolog_term, prolog_theory,
         [atomic << ground << nonvar]).

function(entity_type, prolog_theory,
         prolog_term x prolog_term ---> boolean, [prolog_pred]).
% This predicate is for type checking only, and so the (SICStus)
% Prolog program should include a declaration
%   :- wait entity_type/2.
% to prevent it generating bindings for entity variables.

function(type_attribute, prolog_theory,
         prolog_term x prolog_term ---> boolean, [prolog_pred]).

fitting_morph(pt_id, [sort(prolog_term is prolog_term),
                      mode(atomic of prolog_term is
                           atomic of prolog_term   ),
                      mode(ground of prolog_term is
                           ground of prolog_term   ),
                      mode(nonvar of prolog_term is
                           nonvar of prolog_term   ),
                      op(entity_type is entity_type),
                      op(type_attribute is type_attribute)]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory pair(TA, TB)  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

theory(pair(_TA:triv,_TB:triv), []).

new_sort(pair(_, _), pair, [unifiable, nonvar]).

submodes(pair, pair(_,_), [unifiable << nonvar]).

unification_mode(pair, pair(_,_), unifiable).

function(pair, pair(TA, TB),
         elt of TA  x  elt of TB ---> pair, []).

mode_rank(pair, pair(TA, TB),
```

```
          elt of TA x elt of TB ---> pair,
          u_mode(elt of TA) x u_mode(elt of TB) ---> u_mode(pair)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory triple(TA, TB, TC)  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

theory(triple(_TA:triv,_TB:triv,_TC:triv), []).

new_sort(triple(_,_,_), triple, [unifiable, nonvar]).
submodes(triple, triple(_,_,_), [unifiable << nonvar]).

unification_mode(triple, triple(_,_,_), unifiable).

function(triple, triple(TA, TB, TC),
         elt of TA  x  elt of TB  x  elt of TC  --->  triple,
         []).

mode_rank(triple, triple(TA, TB, TC),
          elt of TA x elt of TB x elt of TC ---> triple,
          u_mode(elt of TA) x u_mode(elt of TB) x u_mode(elt of TC)
              ---> u_mode(triple)
         ).


%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory bag(T)  %%%
%%%%%%%%%%%%%%%%%%%%%%

theory(bag(_T : triv), []).

new_sort(bag(_), bag, [head_enum_bag]).

unification_mode(bag, bag(_), head_enum_bag).

function({}, bag(_), ---> bag, []).

function(::, bag(T), elt of T  x  bag  --->  bag,
         [left_commutative]).

mode_rank({}, bag(_), ---> bag, ---> head_enum_bag).
mode_rank(::, bag(T),
          elt of T  x  bag  --->  bag,
          any of u  x  any of u  --->  head_enum_bag).
% To have mode head_enum_bag, the 2nd arg. of a term A::B must
% be a variable or have mode head_enum_bag (these are the only
```

```
% terms in mode any)

function(u, bag(_T), bag x bag ---> bag, [ac]).

mode_rank(u, bag(_), bag x bag ---> bag,
          any of u  x  any of u  --->  any of u).

rewrite_rule(bag(_), u, {} u X ==> X, true).
rewrite_rule(bag(_), u, A::X u Y ==> A::(X u Y), true).

function(\, bag(_), bag x bag ---> bag, []).

mode_rank(\, bag(_), bag x bag ---> bag,
          any of u  x  any of u --->  any of u).

rewrite_rule(bag(_), \, X \ {} ==> X, true).
rewrite_rule(bag(_), \, {} \ _ ==> {}, true).
rewrite_rule(bag(_), \, A::X \ A::Y ==> X\Y, true).
rewrite_rule(bag(_), \, A::X \ Y  ==>  A::(X\Y), not_in(A,Y)).
rewrite_rule(bag(_), \, X \ A::Y ==> X\Y, not_in(A,X)).
rewrite_rule(bag(_), \, (X \ A::Y)\Z ==> (X \ Y)\A::Z, true).

function(=<, bag(_), bag x bag ---> boolean, []).
% '=<' is a sub-bag predicate

rewrite_rule(bag(_), =, X\Y = {}  ==>  X =< Y, true).

rewrite_rule(bag(_), =<, {} =< X  ==>  true, true).
rewrite_rule(bag(_), =<, A::X =< A::Y  ==>  X =< Y, true).
rewrite_rule(bag(_), =<, X =< A::Y  ==>  X =< Y, not_in(A,X)).
rewrite_rule(bag(_), =<,
             A::X =< Y  ==>  Y = A::Z  &  X =< Z, true).
rewrite_rule(bag(_), =<,
             A::X \ Y =< Z  ==>  Y = A::W  &  X\W =< Z,
             not_in(A,Z)                                  ).
rewrite_rule(bag(_), =<,
             X u Y =< Z  ==>  X =< Z  &  Y =< Z, true).

function(not_in, bag(T), elt of T x bag ---> boolean, []).

rewrite_rule(bag(_), not_in, not_in(A, {}) ==> true, true).
rewrite_rule(bag(_), not_in, not_in(A, A::X) ==> false, true).
rewrite_rule(bag(_), not_in, not_in(A, B::X) ==> not_in(A, X),
             A \= B                                          ).

rewrite_rule(bag(_), =, A::X = A::Y  ==>  X = Y, true).
% avoid unification where possible
```

```
%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory entity  %%%
%%%%%%%%%%%%%%%%%%%%%%%%

theory(entity(PT:prolog_theory), [protecting(PT)]).

new_sort(entity(_), entity, [ent_u_mode]).

unification_mode(entity, entity(_), ent_u_mode).

function($, entity(_),
         prolog_term x prolog_term ---> entity, []).
% terms 'name$type' are used to represent typed entity variables
% in the planner.

mode_rank($, entity(_),
          prolog_term x prolog_term ---> entity,
          any of u  x  any of u ---> ent_u_mode     ).

fitting_morph(ent_as_triv, [sort(elt is entity),
                            mode(u_mode(elt) is
                                 ent_u_mode of entity)]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory attribute  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

theory(attribute(PT:prolog_theory), [protecting(PT)]).

new_sort(attribute(_), attribute, [ground]).

unification_mode(attribute, attribute(_), ground).

function(att, attribute(_), prolog_term ---> attribute, []).

mode_rank(att, attribute(_),
          prolog_term ---> attribute, any of u ---> any of u).

fitting_morph(att_as_triv, [sort(elt is attribute),
                            mode(u_mode(elt) is
                                   ground of attribute)]).

%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory value  %%%
%%%%%%%%%%%%%%%%%%%%%%%
```

```
theory(value(PT:prolog_theory), [protecting(PT)]).

new_sort(value(_), value, []).

function(val, value(_), prolog_term ---> value, []).

mode_rank(val, value(_),
          prolog_term ---> value, any of u ---> any of u).
fitting_morph(val_as_triv, [sort(elt is value),
                            mode(u_mode(elt) is any of u)]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory statespec_entry  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

theory(statespec_entry(PT:prolog_theory),
       [protecting(pt = PT),
        protecting(triple(entity(pt / pt_id)    / ent_as_triv,
                          attribute(pt / pt_id) / att_as_triv,
                          value(pt / pt_id)     / val_as_triv )
                    * [sort(triple to err_entry)]              )]).

new_sort(statespec_entry(_), entry, []).

subsorts(statespec_entry(_), [entry << err_entry]).

sort_constraint(statespec_entry(_), triple,
                entity x attribute x value ---> entry,
                triple(E, A, _),
                types_ok(E, A)                          ).

function(types_ok, statespec_entry(_),
         entity x attribute ---> boolean, []).

rewrite_rule(statespec_entry(_), types_ok,
             types_ok(Entity$Type, att(Attribute)) ==>
                 type_attribute(Type, Attribute) &
                 entity_type(Type, Entity)              ,
             true                                        ).


%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Theory statespec  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
theory(statespec(PT:prolog_theory),
       [protecting(basic_statespec =
                     bag(statespec_entry(PT / pt_id)
                          / [sort(elt is err_entry),
                             mode(u_mode(elt) is
                                    unifiable of err_entry)])
                      * [sort(bag to statespec),
                         mode(head_enum_bag of bag  to
                              head_enum_statespec     )]       ),
        protecting(pair(univ / univ_as_triv, univ / univ_as_triv))
       ]).

new_sort(statespec(_), consistent_spec, []).

subsorts(statespec(_), [consistent_spec << statespec]).

function({} of basic_statespec, statespec(_),
         ---> consistent_spec, []           ).

function(\ of basic_statespec, statespec(_),
         consistent_spec x statespec ---> consistent_spec, []).

function(\\, statespec(_),
         consistent_spec x statespec ---> consistent_spec, []).

mode_rank(\\, statespec(_),
          consistent_spec x statespec ---> consistent_spec,
          any of u  x  any of u  --->  any of u            ).

rewrite_rule(statespec(_), \\, X \\ {} ==> X, true).
rewrite_rule(statespec(_), \\, {} \\ _ ==> {}, true).
rewrite_rule(statespec(_), \\,
             triple(E,A,V)::X \\ triple(E,A,_)::Y  ==>  X \\ Y,
             true                                              ).
rewrite_rule(statespec(_), \\,
             triple(E,A,V)::X \\ Y  ==>
                triple(E,A,V)::(X\\Y)  , no_entry(E,A,Y)).
rewrite_rule(statespec(_), \\,
             X \\ triple(E,A,_)::Y  ==>  X \\ Y,
             no_entry(E,A,X)                   ).
rewrite_rule(statespec(_), \\,
             (X \\ A::Y)\\Z ==> (X \\ Y) \\ A::Z, true).


function(=< of basic_statespec, statespec(_),
         consistent_spec x consistent_spec ---> consistent_spec).
```

```
sort_constraint(statespec(_), ::,
                entry x consistent_spec ---> consistent_spec,
                Triple :: S,
                new_assignment_triple_ok(Triple, S)          ).

function(new_assignment_triple_ok, statespec(_),
        entry x statespec ---> boolean, []    ).

rewrite_rule(statespec(_), new_assignment_triple_ok,
            new_assignment_triple_ok(_T, {}) ==> true,
            true).
rewrite_rule(statespec(_), new_assignment_triple_ok,
            new_assignment_triple_ok(triple(E1, A1, V),
                                     triple(E2, A2, _) :: S)
              ==> pair(E1, A1) \= pair(E2, A2)  &
                  new_assignment_triple_ok(triple(E1, A1, V), S),
            true                                                ).

function(no_entry, statespec(_),
        entity x attribute x consistent_spec ---> boolean, []).

function(update, statespec(_),
        consistent_spec x consistent_spec ---> consistent_spec, []).

mode_rank(update, statespec(_),
          consistent_spec x consistent_spec ---> consistent_spec,
          any of u  x  any of u  --->  any of u                  ).

rewrite_rule(statespec(_), update, update({}, X) ==> X, true).
rewrite_rule(statespec(_), update, update(X, {}) ==> X, true).
rewrite_rule(statespec(_), update,
            update(X, A::Y) ==> A::update(X \\ A::{}, Y), true).

rewrite_rule(statespec(_), no_entry,
            no_entry(_, _, {}) ==> true, true).
rewrite_rule(statespec(_), no_entry,
            no_entry(E, A, triple(E,A,_)::X) ==> false, true).
rewrite_rule(statespec(_), no_entry,
            no_entry(E, A, triple(E2,A2,_)::X)
              ==> no_entry(E, A, X),
            pair(E,A) \= pair(E2,A2)          ).

rewrite_rule(statespec(_), =,
            ((A::X) as consistent_spec) = Y\B::Z  ==>
                 A::W = Y\Z &  X = W\B::{}          , true).
rewrite_rule(statespec(_), =,
            triple(E1,A1,V1)::X = Y \\ triple(E2,A2,V2)::Z  ==>
```

```
                triple(E1,A1,V1)::W = Y\\Z  &
                X = W\\triple(E2,A2,V2)::{}  ,
             true                                                    ).
rewrite_rule(statespec(_), =,
             triple(E,A,V1)::X = update(triple(E,A,V2)::Y, Z)  ==>
                Z = triple(E,A,V1)::W  &  X = update(Y,W)          ,
             V1 \= V2                                                ).
rewrite_rule(statespec(_), =,
             triple(E,A,V)::X = update(Y, Z)  ==>
                Z = triple(E,A,V)::W  &  X = update(Y,W),
             no_entry(E,A,Y)                                  ).
rewrite_rule(statespec(_), =,
             X = update(X, Y)  ==>  Y =< X, true).

rewrite_rule(statespec(_), =< of enrichment,
             X \ triple(E,A,V)::Y  =<  Z
                 ==>  (X\Y =< triple(E,A,V)::Z) of enrichment,
             no_entry(E,A,Z)                                      ).
```

Prolog Operator Declarations

```
% Operators for keywords, etc.

:- op(950, xfx, [ ==> ]).
:- op(700, xfx, [ is ]).   % 'is' is predefined
:- op(700, xfx, [ --->, to, from ]).
:- op(700,  fx, [ ---> ]).
:- op(699, xfy, [ x ]).
:- op(698, xfx, [ as ]).
:- op(697, xfy, [ of ]).
:- op(400, xfy, [ << ]).   % Changed from predefined type of 'yfx'.
:- op(0,    fx, [(mode)]). % Cancel built-in high priority
                           %  prefix property.

% Operators for defined functions:

:- op(900, xfy, [ & ]).
/* op(700, xfx, [ =, ==, =< ].     -- predefined -- */
:- op(700, xfx, [ \= ]).
:- op(650, yfx, [ u, \, \\ ]).
:- op(600, xfy, [ :: ]).
:- op(600, xfx, [ $ ]).
```

Appendix C

Combining Equational Unification and Matching Algorithms

Reasoning about equality, substitutions and unification can be rather awkward because of the notational and technical intricacies of the standard theoretical treatment (such as in [Huet & Oppen 80]). In particular, when generating substitutions (e.g. for matching or unifying terms) it is usually necessary to restrict the variables that can appear within the image of any variable that is bound by the substitution. This prevents two distinct variables becoming accidentally and unintentionally unified. Other complications include the necessity of distinguishing between the distinct occurrences of a variable or subterm within a term.

In order to abstract away from these details, Rydeheard and Stell (1987) propose an alternative treatment of equational deduction using category theory. This approach concentrates on the compositional properties of substitutions, rewrite rules, etc. without worrying about details of representation or implementation. Another advantage is that category theory is generally constructive by nature: proofs often involve the construction of objects from other objects using standard techniques, and these transform quite readily to abstract algorithms that are independent of any particular data structures. In this appendix we present a categorical description of Yelick's procedure for combining unification algorithms (see Section 6.2.3.1) based on the outline presented in [Rydeheard & Stell 87], and use this to develop a similar algorithm for matching terms in a combination of disjoint collapse-free and regular equational theories.

It is beyond the scope of this thesis to discuss category theory in any depth — in this appendix it is used only for purposes of illustration and motivation. We therefore present only the bare essentials required to understand the following discussion. For a deeper coverage of category theory see [Pierce 90, Goldblatt 79, Arbib & Manes 75, MacLane 71].

A category is an abstract structure consisting of a set of objects together with a set of *arrows* (also called *morphisms*) which represent some type of (directed) relationship between two objects. Examples of categories include sets together with functions on sets, partial orders (where the arrows represent the ordering relationships between objects), and logical formulae together with proofs demonstrating how one formula follows from another. An arrow f from A to B is written as $f: A \longrightarrow B$ or $A \xrightarrow{f} B$. The objects A and B are considered to be part of the arrow f, and so, for example, in the category of sets with functions, the unique function $f: \{a\} \longrightarrow \{b\}$ constitutes a different arrow when it is regarded as a function from $\{a\}$ to $\{b, c\}$.

In Rydeheard and Stell's treatment of unification, the objects are sets (of variables) and an arrow $f: A \longrightarrow B$ is a substitution mapping variables of A to terms in $T_\Sigma(B)$, i.e. the set of terms formed using the function symbols of the equational theory (with signature Σ) and variables from B. A unification problem involving terms with variables from the set X is defined as a set of equations $\{s_i = t_i \mid i \in Z\}$ where Z is an index set of 'place-holder' variables. These equations are represented as the parallel pair of arrows $Z \underset{g}{\overset{f}{\rightrightarrows}} X$ where f and g are the substitutions defined by $f(i) = s_i$ and $g(i) = t_i$. Solving this set of equations involves finding a substitution $q: X \longrightarrow Y$ for some set Y such that $f.q = g.q$ (where '.' denotes the composition of arrows, defined so that the ordering along the direction of the arrows is preserved, i.e. $f.g = g \circ f$). In category theory, statements of equality such as this are represented by 'commutative diagrams' which assert that all compositions of arrows that form a path between the same two objects are equal. The exception to this rule is that a parallel pair of arrows $A \underset{g}{\overset{f}{\rightrightarrows}} B$ are not considered equal — they only serve to form part of two longer paths which the diagram states to be equal. Thus, the equality $f.q = g.q$ is represented by the diagram $Z \underset{g}{\overset{f}{\rightrightarrows}} X \xrightarrow{q} Y$. In the standard form of the unification problem, Z is a singleton set $\{z\}$ and this diagram asserts that q is a unifier of the terms $f(z)$ and $g(z)$

(these represent the image of z under f and g respectively; f and g are not function symbols of the equational theory).

In the empty equational theory, there is a unique most general unifier of f and g. In category theory, this corresponds to the notion of a *coequalizer* of f and g. Rydeheard and Stell show how the unification algorithm of [Martelli & Montanari 82] (in its nondeterministic, unoptimised form) can be derived as a recursive process of constructing coequalizers from other coequalizers. For the general case of equational unification, equality relationships between terms (and in particular rewrite rules) can be represented by *2-cells*—arrows that map between pairs of arrows with the same endpoints. These 2-cells generate an equivalence relation $\equiv$ on arrows (i.e. substitutions) and this induces a *quotient functor* which maps sets and substitutions into their counterparts in a new category in which the equality of arrows is defined by the relation $\equiv$. The technical details are not important to this discussion; it is sufficient to note that in the following diagrams, two arrows $f, g: A \longrightarrow B$ are considered equal if $\forall a \in A\ f(a) =_T g(a)$, where $=_T$ is the equational theory for which the unification algorithm is being derived.

For equational unification, there may not be a single most general unifier, and the notion of a coequalizer is replaced by that of a *complete set of solutions*: an arrow q such that $Z \overset{f}{\underset{g}{\rightrightarrows}} X \xrightarrow{q} Y$ commutes in the quotient category is said to be a *solution* of f, g, and a set of solutions for a problem f, g is *complete* if any solution q' can be expressed as a composition $q.r$ for some q in the set.

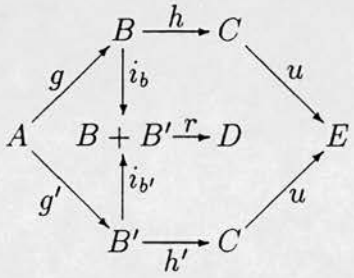
C.1 Deriving Yelick's Unification Procedure: A Categorical View

The reader is referred to Section 6.2.3.1 for an overview of Yelick's procedure.

The unification of two terms $t, t' \in T_\Sigma(C)$ is modelled as the problem of finding a complete set of solutions u for f, f' in the following diagram:

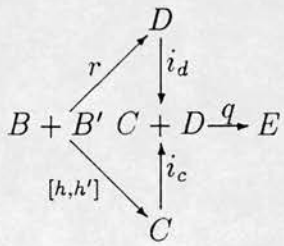
$$A \overset{f}{\underset{f'}{\rightrightarrows}} C \xrightarrow{u} E$$

where A is a singleton $\{a\}$, $f(a) = t_1$ and $f'(a) = t_2$. If t_1 and t_2 belong to the same subtheory (T_i say), they can be homogenised to give terms $\hat{t}$ and $\hat{t}'$ with preserving substitutions h and h' respectively. The unification algorithm for the subtheory T_i can then be used to find a unifier r for $\hat{t}$ and $\hat{t}'$. This is represented by the diagram:



where $B = Dom(h)$, $B' = Dom(h')$, and $B + B'$ is the *co-product* of B and B', which in this category corresponds to their disjoint union, with i_b and $i_{b'}$ being the corresponding injection functions. r is a solution of the problem $A \underset{g'.i_{b'}}{\overset{g.i_b}{\rightrightarrows}} B + B' \xrightarrow{r} D$. This does not correspond exactly to Yelick's algorithm, where the homogeneous terms may have some variables in common (if there are occurrences of a variable in both t and t' which are not within some subterm that is replaced during homogenisation). These will not be treated as distinct variables during unification in the subtheory, as the above diagram suggests. However, this difference is not significant.

Given r, Rydeheard and Stell state that a solution to the problem is given by $u = i_c.q$ for any q that makes the following diagram commute[1]:



(I)

where $[h, h']$ is the unique extension of h and h' to $B + B'$ — this is the union of the two preserving substitutions. $i_c.q$ represents the restriction of q to C. Solving this problem amounts to 'unifying' the substitutions r and $[h, h']$. Yelick achieves this with her *map_unify* procedure, which unifies the images under the two substitutions of each variable in their combined domains in turn. In the categorical treatment, this

[1] although no further details are given. The material in the remainder of this appendix is the author's own work.

corresponds to recursively solving the unification problem

$$\{b_1\} \xrightarrow{i_1} B + B' \underset{[h,h'].i_c}{\overset{r.i_d}{\rightrightarrows}} C + D \xrightarrow{q_1} E_1$$

to find q_1 (where $B + B' = \{b_1, \ldots, b_n\}$), followed by another call to *map_unify* to find q_2 such that the following diagram commutes:

$$\{b_2, \ldots, b_n\} \xrightarrow{i_2} B + B' \underset{[h,h'].i_c}{\overset{r.i_d}{\rightrightarrows}} C + D \xrightarrow{q_1} E_1 \xrightarrow{q_2} E_2$$

These two diagrams can be combined to give:

$$B + B' \underset{[h,h']}{\overset{r}{\rightrightarrows}} C + D \xrightarrow{q_1.q_2} E_2$$

and this shows that putting $q = q_1.q_2$ in Diagram (I) above gives $i_c.q_1.q_2$ as a unifier for t and t'.

This decomposition is the essence of Yelick's *cr_unify* procedure, and defines a nondeterministic form of the algorithm which generates a single unifier at a time. Figure C.1 presents the algorithm in this form (for illustrative purposes), using a Prolog-like language enhanced with functional notation to allow the concise expression of operations involving substitutions and the inner structure of (order-sorted) terms. The recursive decomposition step is implemented by the predicate cr_unify_nonvars/3. All possible unifiers will be generated by backtracking (although for the planning system discussed in this thesis it is desirable to collect the complete set of unifiers, which of course assumes that only finitary theories will be used). The version of the algorithm presented here is partly based on that of Zachary (1987).

Although the categorical viewpoint gives an elegant description of the recursive structure of this algorithm, it is not so useful for considering the question of termination. In particular, Yelick's algorithm includes a test that is not obvious from the discussion above. In the case of unifying a variable v and a term t, it is necessary to check that v does not occur within a subterm of t whose principal functor belongs to a different subtheory from that of t (hence the test $V \notin VCod(\theta)$ in cr_unify_var_nonvar/3). Because of the restriction to regular theories these two terms are not unifiable; however

```
cr_unify(T1, T2, μ) :-
    ( var(T1)  ->
          ( var(T2)  ->  μ = {T2/T1}
          ; cr_var_nonvar_unify(T1, T2, μ)   )
    ; /* non_var(T1) */
          ( var(T2)  ->  cr_var_nonvar_unify(T2, T1, μ)
          ; cr_unify_nonvars(T1, T2, μ)                    )
    ).

cr_unify_var_nonvar(V, T, μ) :-
    homogenise(T, H, θ),        % θ is the preserving substitution.
    V ∉ VCod(θ),
    ( V ∉ vars(H)  ->  μ = {T/V}
    ; unify_in_sub_theory(functor_theory(T), V, H, ρ),
      map_unify(ρ, θ, Dom(ρ) ∪ Dom(θ), μ)
    ).

cr_unify_nonvars(T1, T2, μ) :-
    functor(T1) = functor(T2),
    homogenise(T1, H1, θ1),
    homogenise(T2, H2, θ2),
    sub_theory_unify(functor_theory(T1), H1, H2, ρ),
    θ = θ1 ∪ θ2,
    map_unify(ρ, θ, Dom(ρ) ∪ Dom(θ), μ).

map_unify(σ, τ, Var_Set, μ) :-
    ( choose_var(Var_Set, V) ->
          cr_unify(σV, τV, π1),
          map_unify(ρ ∘ σ, ρ ∘ τ, Var_Set − V, π2),
          μ = π2 ∘ π1
    ; μ = {}
    ).
```

Greek letters (σ, τ, ...) are variables denoting substitutions, and functional notation is used to express the operations of substitution application (σV), union ($\cup$) and composition ($\circ$), and also for the following constructs:

- $\{T/V\}$ denotes the singleton substitution mapping V to T. $\{\}$ is the identity substitution.
- $Dom(\sigma) = \{x \in \mathbf{X} \mid \sigma x \neq x\}$, where $\mathbf{X}$ is the set of all variables.
- $VCod(\sigma) = vars(\{\sigma x \mid x \in Dom(\sigma)\})$, i.e. the set of variables introduced by σ.
- *functor_theory*(T) is the theory to which *functor*(T), the principal functor of T, belongs.

Figure C.1: The *cr_unify* procedure

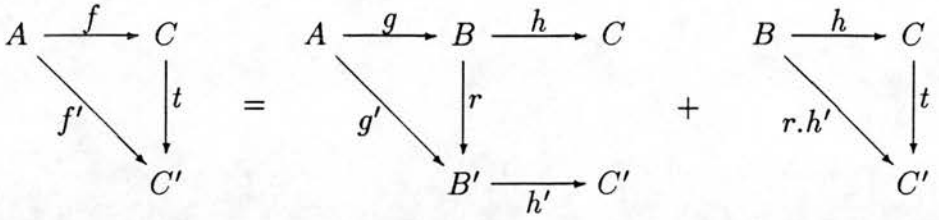
if this test is not included the procedure may loop. This is a weakened version of the 'occurs check' in the standard unification algorithm, but here it cannot be applied to homogeneous variables as within any of the equational subtheories, a variable may be unifiable with a term containing itself (e.g. if there exists an axiom $f(f(x)) = f(x)$ then $\{f(y)/x\}$ unifies x and $f(x)$).

Zachary extends this algorithm to deal with moded unification procedures. This requires two additions: cr_unify/3 should initially test to see if T_1 and T_2 satisfy the unification mode restriction for their sort (or greatest common sort when extended to order-sorted algebra), signalling a mode failure if this test fails. Also, map_unify/4 must take the mode restrictions into account when selecting a variable from Var_Set, delaying the consideration of any variable for which the unification of σV and τV is not well-moded. If no suitable variable is found, a mode failure is signalled. In our system it is more useful to collect these ill-moded 'unification literals' together with the substitutions corresponding to the solved part of the unification problem. They can then be added to the list of current goals when the associated partial unifier is selected. This approach was not possible in Zachary's language Denali because of its strict enforcement of data abstraction principles (see Section 7.2.2).

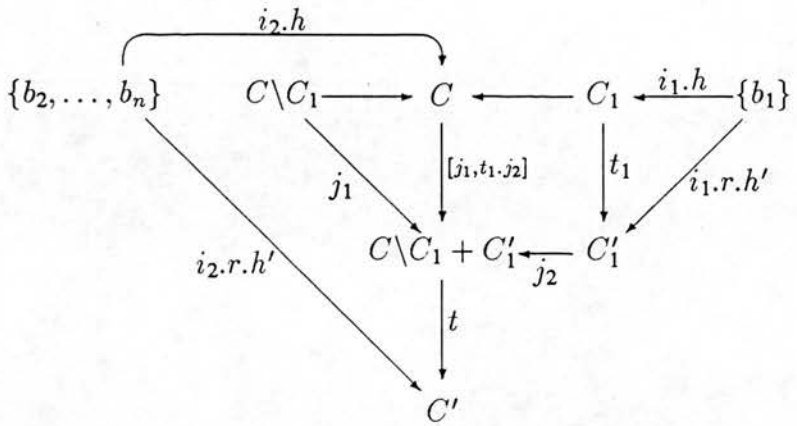
C.2 A Procedure for Combining Equational Matching Algorithms

The planning framework proposed in this thesis uses an implementation of state specifications using equational logic, with the equations of the theory being partitioned into a set of confluent and terminating rewrite rules R, and a disjoint union of collapse-free regular theories E for which unification algorithms have been implemented. The matching of terms to the left hand sides of rewrite rules must be performed with respect to the theory E. This requires an algorithm for equational matching analogous to Yelick's combined unification procedure.

Rydeheard and Stell's categorical treatment of unification provides a useful tool for developing a combined matching algorithm. A recursive procedure for matching can be derived from the following decomposition of the matching problem:



This states that the matching of two terms can be achieved by first matching the homogenised terms, with any resulting matchers r being used in the recursive process of 'matching' the substitutions h and $r.h'$. A counterpart to the *map_unify* procedure is suggested by the following diagram:



where $B = \{b_1, \ldots, b_n\}$, $C_1 = VCod(h\restriction_{\{b_1\}})$ (i.e. the set of variables appearing in the image of b_1 under h), $C'_1 = VCod(t_1\restriction_{C_1})$, i_1 and i_2 are the inclusions of $\{b_1\}$ and $\{b_2, \ldots, b_n\}$ into B, and j_1, j_2 and the two unlabelled arrows are the pairs of inclusions into the coproducts (disjoint unions) $C \backslash C_1 + C'_1$ and C respectively.

This diagram states that a matching substitution t can be constructed by choosing a variable b_1, matching its images under h and $r.h'$ to get a substitution t_1 and then recursively solving the problem of matching the substitutions $t_1 \circ h$ and $h' \circ r$ for the remaining variables $\{b_2, \ldots, b_n\}$. This leads to the *cr_match* algorithm shown in Figure C.2. The decomposition used in this algorithm has the advantage over other possible decompositions of not requiring the implementation of algorithms to test for equality in the various equational subtheories, in addition to the specialised matching algorithms that must be provided.

The restricted occurs check $V \notin VCod(\theta)$ is required to prevent looping, and as a result, this is really an algorithm for *semi-unification* ([Huet 76], discussed in [Bürckert 86]) rather than matching. Semi-unification is the problem of finding unifiers

```
cr_match(T1, T2, μ) :-
    ( var(T1)  ->
        ( var(T2)  ->  μ = {T2/T1}
        ; cr_var_nonvar_match(T1, T2, μ)   )
    ; /* non_var(T1) */
        ( non_var(T2)  -> cr_match_nonvars(T1, T2, μ)
          % otherwise fail (assuming there are no collapse equations)
        )
    ).

cr_match_var_nonvar(V, T, μ) :-
    homogenise(T, H, θ),      % θ is the preserving substitution.
    V ∉ VCod(θ),
    ( V ∉ vars(H)  ->  μ = {T/V}
    ; match_in_sub_theory(functor_theory(T), V, H, ρ),
      map_match({}, θ ∘ ρ, Dom(θ), μ)
    ).


cr_match_nonvars(T1, T2, μ) :-
    functor(T1) = functor(T2),
    homogenise(T1, H1, θ1),
    homogenise(T2, H2, θ2),
    sub_theory_match(functor_theory(T1), H1, H2, ρ),
    map_match(θ1, θ2 ∘ ρ, Dom(θ1), μ).

map_match(σ, τ, Var_Set, μ) :-
    ( choose_var(Var_Set, V) ->
        cr_match(σV, τV, π1),
        map_match(π1 ∘ σ, τ, Var_Set − V, π2),
        μ = π2 ∘ π1
    ; μ = {}
    ).
```

Figure C.2: The *cr_match* procedure

μ of two terms s and t with $Dom(\mu) \subseteq vars(s) \setminus vars(t)$, whereas matching is sometimes defined as the problem of finding μ such that $t = \mu s$, and this allows (for example) the substitution $\{f(x)/x\}$ to be considered to match x to $f(x)$. In practice, however, this distinction is not important.

In hindsight, the *cr_match* algorithm appears to be a fairly straightforward modification of Yelick's unification procedure; however, the author found that the categorical viewpoint greatly clarified exactly *which* straightforward modification was appropriate, and for this reason, the category theory motivation is presented here. No attempt has yet been made to prove the completeness or termination of this algorithm.

Note that unlike our planning system's use of equational unification, there is no need to find complete sets of matchers. The equational matching algorithm is intended to be used for matching terms to the left hand sides of rewrite rules. Once a single matching substitution has been found, the rewrite rule can be applied, and as long as the set of rewrite rules is confluent, no possible solutions will be lost by ignoring any other possible matchers.

Instead of implementing a specialised matching algorithm for disjoint unions of regular collapse-free theories, an alternative approach to equational matching is presented by Bürckert (1986). Bürckert's algorithm computes a set of most general *restricted* unifiers of two terms (i.e. with domains restricted to some subset of the variables in the two terms) from the set of their most general unifiers. The algorithm can be applied in theories that are *almost collapse-free* — theories in which the only collapse equations that appear have the form $f(x_1, \ldots, x_n) = x_i$ for some i, $1 \leq i \leq n$ where the x_i are distinct variables. This includes the class of theories for which Yelick's unification procedure works. Computing restricted unifiers in this way is equivalent to unifying terms in a new theory obtained by enriching the original one with additional constant symbols, and Bürckert gives an example showing that adding constants to a theory that is not almost collapse-free can turn a decidable unification problem into an undecidable one. This technique may not be practically useful, as various authors (e.g. [Gramlich & Denzinger 88]) have remarked that using unification algorithms to perform matching has been found to be too expensive in practice.

Appendix E

Current State of the Implementation

At the time of writing, most of the implementation effort has been focused on the preprocessing component which reads in and parses the equational theory presentation for the state specification data structure, inferring sorts for the variables appearing in the rewrite rules and checking the various conditions outlined in Chapter 7. This results in a 'compiled' representation of the state specification equational theory which uses data structures designed to allow the various plan-time computations to be performed efficiently.

The planning component currently consists of a set of predicates for parsing and pretty-printing terms, performing order-sorted equational unification and matching, the rewriting of terms, and mode checking. This allows experimental planners to be written as simple Prolog programs which read in the compiled theory and then repeatedly alternate the normalisation of the current subgoals with the selection and (reverse) application of inference rules, calling upon these predicates to perform the necessary low-level computations. At present, three aspects of the proposed framework are not yet implemented (although all are straightforward to add): Prolog goals from the atomic action *PlanSpec* axioms cannot be evaluated as subgoals of the current proof state; the need to test sort constraint conditions (resulting from variable instantiations making terms become possibly ill-sorted) is not detected; and ill-moded unification subproblems are not postponed until sufficiently instantiated (Zachary's approach of signalling a mode failure is still followed, although the alternative approach can be simulated by decomposing the equality constraints and relying on a fixed order of evaluation so that $A : X = B : (Y \backslash Z)$ becomes $A : X = B : W \wedge W = Y \backslash Z$ for example).

Appendix D

Predicate Descriptions for the Soma-World Example

This appendix describes the intended meaning of the predicates appearing in the plan specifications for the Soma-world behavioural operators shown in Figure 4.2. Note also that in this figure, plan specification formulae $\Gamma \triangleright \langle \phi, P, \psi \rangle$ are represented in the diagrammatic form $\phi \overset{P}{\underset{\Gamma}{\Rightarrow}} \psi$, and functional notation is used for the operations of set union ($\cup$) and difference ($\setminus$).

`possible_get_grip(Grip,Config)`

`Grip` is a possible grip for picking up a part in configuration `Config`.

`normalised(Config1,Grip,Config2,Grip2)`

`Config2` is the normalised form of `Config1` and `Grip2` is the result of applying the resulting normalising translation to `Grip1` (the normal form of a configuration has all cubie offsets non-negative and as small as possible).

`fits(Part,Config)`

The set of cubie offsets `Config` is a possible configuration for `Part`.

`supported_by(Config,Occupied_Cubicles)`

All cubie coordinates in `Config` are directly above some cubie position in `Occupied_Cubicles`.

`gripper_clearance_ok(Grip, Config, Occupied_Cubicles)`

There are no cubie coordinates in `Occupied_Cubicles` that will be at a higher level of the assemblage than the gripped cubie of `Grip` when the gripped part is added to the assemblage in configuration `Config` (this prevents collisions between the top of the gripper and other parts in the assemblage).

`finger_clearance_ok(Grip, Config, Occupied_Cubicles)`

There is sufficient room in the assemblage represented by `Occupied_Cubicles` to allow a part to be inserted in configuration `Config` while held with grip `Grip`.

`config_matching_transformation(Config1, Config2, Trans, Rot)`

The transformation represented by the translation `Trans` and rotation `Rot` maps the configuration `Config1` to a configuration that 'matches' `Config2` (taking symmetry into account).

`possible_grip_transformation(Trans, Rot, Grip1, Grip2)`

`Trans` and `Rot` represent a transformation that maps `Grip1` onto `Grip2`.

`possible_rotation(Grip1, Config1, Grip2, Config2)`

There is a possible rotation of the robot end effector that will leave the robot holding a part in configuration `Config2` with grip `Grip2` when performed whilst initially holding the part in configuration `Config1` with grip `Grip1` (this rotation will be determined by the robot controller at run time, and so does not explicitly appear as an argument to this predicate).

`stable(Config, Base)`

A part in configuration `Config` is stable when the (minimal) set of bottom-level cubies `Base` is supported from beneath.

`translate_to_fit_table(Base, Trans, Offset)`

The translation `Trans` transforms the set of cubie coordinates `Base` so that they lie within the area of the regrasp table when offset from the local coordinate frame by `Offset` (representing a vector (x, y, z) where $x, y, z \in \{0, \frac{1}{2}\}$).

`translate_gripped_cubie(Grip1,Trans,Grip2)`

`Grip2` is the result of applying the translation `Trans` to `Grip1` (only the gripped cubie component is affected).

`translate(Config1,Trans,Config2)`

`Config2` is the result of applying translation `Trans` to `Config1`.

Bibliography

Agre, P. E. and Chapman, D. (1987), "Pengi: An Implementation of a Theory of Activity", in *Proc. AAAI-87*.

Agre, P. E. and Chapman, D. (1990), "What Are Plans For?", *Robotics and Autonomous Systems* 6, pp. 17–34.

Allen, J. (1984), "Towards a General Theory of Action and Time", *Artificial Intelligence* 23, pp. 123–154.

Aloimonos, J., Weiss, I. and Bandyopadhyay, A. (1987), "Active Vision", in *Proc. 1st International Conf. on Computer Vision*, pp. 35–54.

Arbib, M. A. and Manes, E. (1975), *Arrows, Structures and Functors*, Academic Press.

Barwise, J., ed. (1977), *Handbook of Mathematical Logic*, Studies in Logic and the Foundations of Mathematics , Vol. 90, North Holland.

Bell, C. E. (1985), "Using Temporal Constraints to Restrict Search in a Planner", AIAI-TR-5, AI Applications Institute, University of Edinburgh.

van Benthem, J. (1982), *The Logic of Time*, Reidel, Dordrecht.

Bockmayr, A. (1986), "Conditional rewriting and narrowing as a theoretical framework for logic-functional programming: A survey", Interner Bericht 10/86, Institut für Informatik 1, Universität Karlsruhe.

Brookes, S. D., Roscoe, A. W. and Winskel, G., eds. (1984), *Seminar on Concurrency*, Lecture Notes in Computer Science , No. 197, Springer-Verlag.

Brooks, R. A. (1985), "A Robust Layered Control System for a Mobile Robot", AI Lab. Memo 864, Massachusetts Institute of Technology.

Brooks, R. A. (1986), "Achieving Artificial Intelligence through Building Robots", AI Lab. Memo 899, Massachusetts Institute of Technology.

Brooks, R. A. (1987), "Planning is just a way of avoiding figuring out what to do next", Internal Paper, AI Laboratory, Massachusetts Institute of Technology.

Brooks, R. A. (1989), "A Robot that Walks: Emergent Behavior from a Carefully Evolved Network", *Neural Computation* 1.

Brooks, R. A. (1990), "Elephants Don't Play Chess", *Robotics and Autonomous Systems* 6, pp. 3–15.

Bundy, A. (1983), *The Computer Modelling of Mathematical Reasoning*, Academic Press.

Bundy, A. and Welham, R. (1981), "Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation", *Artificial Intelligence* 16.

Bürckert, H-J. (1986), "Some Relationships between Unification, Restricted Unification and Matching", SEKI report SR-86-05, Fachbereich Informatik, Universität Kaiserslautern.

Burstall, R. M. (1969), "Proving Properties of Programs by Structural Induction", *Computer Journal* 12, pp. 41–48.

Burstall, R. M. and Goguen, J. A. (1980), "The Semantics of Clear, a Specification Language", in *Proc. Advanced Course on Abstract Software Specifications*, Lecture Notes in Computer Science, No. 86, Springer-Verlag, (also Report CSR-65-80, Department of Computer Science, University of Edinburgh).

Burstall, R. M. and Goguen, J. A. (1981), "An Informal Introduction to Specifications using Clear", in *The Correctness Problem in Computer Science*, R.S. Boyer and J. Strother Moore, eds., Academic Press.

Chapman, D. (1987), "Planning for Conjunctive Goals", *Artificial Intelligence* 32, pp. 333–377.

Clark, K. L. (1978), "Negation as Failure", in *Logic and Databases*, H. Gallaire and J. Minker, eds., Plenum Press, pp. 293–322.

Colmerauer, A. (1982), "Prolog-II Manuel de Référence et Modèle Théorique", Groupe Intelligence Artificelle, Université d'Aix-Marseille II.

Conkie, A. and Chongstitvatana, P. (1990), "An Uncalibrated Stereo Visual Servo System", in *Proc. British Machine Vision Conf.* , Oxford, 1990.

Connell, J. H. (1990), "A Colony Architecture for an Artificial Creature", Ph.D. Thesis, Massachusetts Institute of Technology.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panagaden, P., Sasaki, J. T. and Smith, S. F. (1986), *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall.

Davis, M. (1982), *Computability and Unsolvability*, Dover.

Dean, T. (1985), "Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving", Ph.D. Thesis, Yale University.

Drummond, M. E. (1985), "Refining and Extending the Procedural Net", in *Proc. IJCAI 1985*, pp. 1010–1012.

Drummond, M. E. (1986), "A Representation of Action and Belief for Automatic Planning Systems", AIAI-TR-16, AI Applications Institute, University of Edinburgh.

Drummond, M. E. (1989), "Situated Control Rules", in *Proc. 1st International Conf. on Principles of Knowledge Representation and Reasoning* , Toronto, May 1989, Morgan Kaufmann.

van Emden, M. H. and Kowalski, R. A. (1976), "The Semantics of Predicate Logic as a Programming Language", *JACM* 23, pp. 733–742.

Emerson, E. A. and Clarke, E. M. (1982), "Using Branching Time Logic to Synthesize Synchronization Skeletons", *Science of Computer Programming* 2, pp. 241–266.

Emerson, E. A. and Halpern, J. Y. (1984), "'Sometimes' and 'Not Never' Revisited: On Branching vs. Linear Time", TR-84-01, Department of Computer Science, University of Texas at Austin.

Emerson, E. A. and Lei, C. -L. (1985), "Modalities for Model Checking: Branching Time Logic Strikes Back", TR-85-21, Department of Computer Science, University of Texas at Austin.

Emerson, E. A. and Sistla, A. P. (1985), "Deciding Full Branching Time Logic", TR-85-28, Department of Computer Science, University of Texas at Austin.

Enderton, H. B. (1972), *A Mathematical Introduction to Logic*, Academic Press.

Fay, M. (1979), "First-Order Unification in an Equational Theory", in *Proc. 4th Workshop on Automated Deduction, Austin, Texas*, pp. 161–167.

Fikes, R. E. and Nilsson, N. J. (1971), "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence* 2, pp. 189–208.

Fleming, A. (1985), "Analysis of Uncertainties in a Structure of Parts", *Proc. IJCAI 1985*.

Fox, B. R. and Kempf, K. G. (1985a), "Opportunistic Scheduling for Robotic Assembly", *Proc. IEEE Conf. on Robotics and Automation*, St. Louis.

Fox, B. R. and Kempf, K. G. (1985b), "A Representation for Opportunistic Scheduling", *International Journal of Robotics Research, Semi-Annual Conference*, Paris.

Futatsugi, K., Goguen, J. A., Jouannaud, J-P. and Meseguer, J. (1984), "Principles of OBJ2", ICOT Report TR-097, Institute for New Generation Computer Technology, Tokyo, Japan, (Also in *Proc. Symposium on Principles of Programming Languages*, ACM, 1985, pp. 52-66).

Genesereth, M. R. and Nilsson, N. J. (1987), *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann.

Georgeff, M. P. (1987), "Planning", in *Annual Review of Computer Science*, Vol. 2, pp. 359–400, (also in Allen, J., Hendler, J. and Tate, A., *Readings in Planning*, Morgan Kaufmann, 1990).

Georgeff, M. P. and Lansky, A. L. (1986), "Procedural Knowledge", in *Proc. of the IEEE, Special Issue on Knowledge Representation*, pp. 1383–1398.

Georgeff, M. P. and Lansky, A. L. (1987), "Reactive Reasoning and Planning", in *Proc. AAAI-87*.

Goguen, J. A. (1978), "Order-Sorted Algebra", Technical Report 14, UCLA, Computer Science Department.

Goguen, J. A. (1984), "Parameterized Programming", in *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5.

Goguen, J. A. and Burstall, R. M. (1983), "Introducing Institutions", in *Proc. Workshop on Logics of Programs, Carnegie-Mellon University*, E. Clarke and D. Kozen, eds., Lecture Notes in Computer Science, No. 164, Springer-Verlag, pp. 221–256.

Goguen, J. A., Jouannaud, J-P. and Meseguer, J. (1985), "Operational Semantics for Order-Sorted Algebra", in *12th International Colloquium on Automata, Languages and Programming, Nafplion, Greece*, Wilfried Brauer, ed., Lecture Notes in Computer Science, No. 194, Springer-Verlag.

Goguen, J. A. and Meseguer, J. (1981), "Completeness of Many-Sorted Equational Logic", *SIGPLAN Notices* 16, pp. 24–32.

Goguen, J. A. and Meseguer, J. (1986), "Eqlog: Equality, Types, and Generic Modules for Logic Programming", in *Logic Programming: Functions, Relations and Equations*, Doug DeGroot and Gary Lindstrom, eds., Prentice-Hall.

Goguen, J. A. and Meseguer, J. (1987a), "Models and Equality for Logical Programming", in *Proc. 2nd International Joint Conf. on the Theory and Practice of Software Development (TAPSOFT '87)*, Hartmut Ehrig, Giorgio Levi, Robert Kowalski and Ugo Montanari, eds., Lecture Notes in Computer Science, No. 249, Springer-Verlag, (also, Report CSLI-87-91, Center for the Study of Language and Information, Stanford University).

Goguen, J. A. and Meseguer, J. (1987b), "Extensions and Foundations for Object-Oriented Programming", in *Research Directions in Object-Oriented Programming*, Bruce Shriver and Peter Wegner, eds., MIT Press.

Goguen, J. A. and Meseguer, J. (1988), "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations", Technical Report, SRI International, Computer Science Lab.

Goldblatt, R. (1979), *Topoi: The Categorial Analysis of Logic*, Studies in logic and the foundations of mathematics, Vol. 98, North-Holland.

Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979), *Edinburgh LCF: A Mechanised Logic of Computation*, Springer-Verlag.

Gramlich, B. and Denzinger, J. (1988), "Efficient AC-Matching using Constraint Propagation", SEKI Report SR-88-15, Fachbereich Informatik, Universität Kaiserslautern.

Green, C. (1969), "Application of Theorem Proving to Problem Solving", in *Proc. IJCAI 1969*, pp. 219–239, (also in Webber, B.L. and Nilsson, N.J. (eds.), *Readings in Artificial Intelligence*, Morgan Kaufmann, 1981).

Hayes, P. J. (1985), "The Second Naive Physics Manifesto", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann .

Herold, A. (1986), "Combination of Unification Algorithms", in *Proc. 8th Conf. on Automated Deduction*, J. Siekmann, ed., Lecture Notes in Computer Science, No. 230, Springer-Verlag, pp. 450–469.

Hoare, C. A. R. (1969), "An Axiomatic Basis of Computer Programming", *Comm. ACM* 12, pp. 576–583.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall.

Hölldobler, S. (1988), "From Paramodulation to Narrowing", in *Logic Programming* , Proc. of the 5th International Conference and Symposium, Seattle, 1988, Robert A. Kowalski and Kenneth A. Bowen, eds., MIT Press, pp. 327–342.

Hölldobler, S. (1989), *Foundations of Equational Logic Programming*, Lecture Notes in Artificial Intelligence, No. 353, Springer-Verlag.

Homem, J. and de Roever, W. P. (1986), "The Quest Goes On: A Survey of Proof-Systems for Partial Correctness of CSP", in *Current Trends in Concurrency: Overviews and Tutorials*, J. W. de Bakker, W.-P. de Roever and G. Rozenberg, eds., Lecture Notes in Computer Science , No. 224, Springer-Verlag, Berlin.

Homem de Mello, L. S. and Sanderson, A. C. (1986), "And/Or Graph Representation of Assembly Plans", *Proc. AAAI-86*.

Horn, C. (1988), "The NurPRL Proof Development System", Working Paper 214, Department of Artificial Intelligence, University of Edinburgh.

Huet, G. (1976), "Resolution d'Equations dans les Languages d'Ordre 1, 2, ..., ω", Thèse d'Etat, Univ. Paris VIII.

Huet, G. and Oppen, D. C. (1980), "Equations and Rewrite Rules: A Survey", in *Formal Language Theory*, R. V. Book, ed., Academic Press, (also, Report STAN-CS-80-785, Department of Computer Science, Stanford University).

Hughes, G. E. and Cresswell, M. J. (1968), *An Introduction to Modal Logic*, Methuen, London.

Hullot, J. M. (1980), "Canonical Forms and Unification", in *Proc. 5th Conf. on Automated Deduction* , Les Arcs, France, Lecture Notes in Computer Science , No. 87, Springer-Verlag.

Jouannaud, J-P., Kirchner, C. and Kirchner, H. (1983), "Incremental Construction of Unification Algorithms in Equational Theories", in *Proc. 10th International Colloquium on Automata, Languages and Programming, Barcelona*, J. Diaz, ed., Lecture Notes in Computer Science , No. 154, Springer-Verlag, pp. 361–373.

Jouannaud, J-P., Kirchner, C., Kirchner, H. and Mégrelis, A. (1988), "OBJ: Programming with Equalities, Subsorts, Overloading and Parameterization", in *Proc. International Workshop on Algebraic and Logic Programming, 1988*, J. Grabowski, P. Lescanne and W. Wechler, eds., Lecture Notes in Computer Science , No. 343, Springer-Verlag, pp. 41–52.

Kaelbling, L. P. and Rosenschein, S. J. (1990), "Action and Planning in Embedded Agents", *Robotics and Autonomous Systems* 6, pp. 35–48.

Kaplan, S. (1984), "Fair Conditional Term Rewriting Systems: Unification, Termination and Confluence", Technical Report 194, LRI, Univ. Orsay.

Kirchner, C. (1985), "Méthodes et Outils de Conception Systématique d'Algorithmes d'Unification dans les Théories Equationelles", Thèse de Doctorat d'Etat, Université Nancy I.

Kirchner, C., Kirchner, H. and Meseguer, J. (1988), "Operational Semantics of OBJ3", in *Proc. 15th International Colloq. on Automata, Languages and Programming, 1988*, Timo Lepisto and Arto Salomaa, eds., Lecture Notes in Computer Science , No. 317, Springer-Verlag.

Knuth, D. and Bendix, P. (1970), "Simple Word Problems in Universal Algebras", in *Computational Problems in Abstract Algebra*, J. Leech, ed., Pergamon Press, pp. 263–297.

Koutsou, A. (1986), "Planning Motion in Contact to Achieve Parts Mating", Ph.D. Thesis, University of Edinburgh.

Lansky, A. L. (1986), "A Representation of Parallel Activity based on Events, Structure and Causality ", Technical Note 401, Artificial Intelligence Center, SRI International, Menlo Park, California.

Lansky, A. L. (1990), "Localized Representation and Planning", in *Readings in Planning*, James Allen, James Hendler and Austin Tate, eds., Morgan Kaufmann.

Lifschitz, V. (1987), "On the Semantics of STRIPS", in *Proc. 1986 Workshop on Reasoning about Actions and Plans* , Timberline, Oregon.

Loeckx, J. and Sieber, K. (1987), in *The Foundations of Program Verification* (second edition), Wiley-Teubner.

Lucassen, J. M. (1987), "Types and Effects: Towards the Integration of Functional and Imperative Programming", Ph.D. Thesis (Report MIT/LCS/TR-408), Massachusetts Institute of Technology .

Lyons, D. M. (1986), "RS: A Formal Model of Distributed Computation for Sensory-Based Robot Control", Ph.D. Thesis, Department of Computer and Information Science, University of Massachusetts at Amherst.

MacLane, S. (1971), *Categories for the Working Mathematician*, Springer-Verlag.

Malcolm, C. A. (1987), "Planning and Performing the Robotic Assembly of Soma Cube Constructions", M.Sc. Dissertation, Department of Artificial Intelligence, University of Edinburgh.

Malcolm, C. A. and Smithers, T. (1988), "Programming Assembly Robots in terms of Task Achieving Behavioural Modules: First Experimental Results", in *Proc. International Advanced Robotics Program: Second Workshop on Manipulators, Sensors and Steps towards Mobility, Section 15, University of Salford, October 1988*, (also, Research Paper 410, Department of Artificial Intelligence, University of Edinburgh).

Malcolm, C. A. and Smithers, T. (1990), "Symbol Grounding via a Hybrid Architecture in an Autonomous Assembly System", *Robotics and Autonomous Systems* 6, pp. 123–144, (also, Research Paper 420, Department of Artificial Intelligence, University of Edinburgh).

Martelli, A. and Montanari, U. (1982), "An Efficient Unification Algorithm", *ACM Trans. on Prog. Languages and Systems* 4.

McCarthy, J. and Hayes, P. J. (1969), "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in *Machine Intelligence*, Vol. 4, pp. 463–502.

McDermott, D. (1982), "A Temporal Logic for Reasoning about Processes and Plans", *Cognitive Science* 6, pp. 101–155.

Meseguer, J. and Goguen, J. A. (1985), "Initiality, Induction and Computability", in *Algebraic Methods in Semantics*, M. Nivat and J. C. Reynolds, eds., Cambridge University Press, pp. 459–541.

Meseguer, J., Goguen, J. A. and Smolka, G. (1989), "Order-Sorted Unification", *Journal of Symbolic Computation* 8, pp. 383–413, (also, Report CSLI-87-86, Center for the Study of Language and Information, Stanford University).

Milner, A. J. R. G. (1980), *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, No. 92, Springer-Verlag.

Naish, L. (1985), "Automatic Control for Logic Programs", *Journal of Logic Programming* 2, pp. 167–183.

Newell, A. and Simon, H. A. (1963), "GPS, A Program that Simulates Human Thought", in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, eds., R. Oldenbourg KG, pp. 279–293, (also in Allen, J., Hendler, J. and Tate, A., *Readings in Planning*, Morgan Kaufmann, 1990).

Nilsson, N. J. (1984), "Shakey the Robot", Technical Report 323, Artificial Intelligence Center, SRI International, Menlo Park, California.

Nutt, W., Réty, P. and Smolka, G. (1989), "Basic Narrowing Revisited", *Journal of Symbolic Computation* 7, pp. 295–317.

Paulson, L. C. (1983), "A Higher-Order Implementation of Rewriting", *Science of Computer Programming* 3, pp. 119–149.

Paulson, L. C. (1986), "Natural Deduction as Higher Order Resolution", *Journal of Logic Programming* 3, pp. 237–258.

Paulson, L. C. (1987a), *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge University Press.

Paulson, L. C. (1987b), "The Representation of Logics in Higher-Order Logic", Technical Report 113, Computer Laboratory, University of Cambridge.

Pierce, B. C. (1990), "A Taste of Category Theory for Computer Scientists", Report CMU-CS-90-113 (Revised version of CMU-CS-88-203), Department of Computer Science, Carnegie-Mellon University, (To appear in *Computing Surveys*)..

Popplestone, R. J., Ambler, A. P. and Bellos, I. M. (1979), "An Interpreter for a Language Describing Assemblies", *Artificial Intelligence* 13.

Pratt, V. (1976), "Semantic Considerations on Floyd-Hoare Logic", in *Proc. 17th IEEE Symposium on the Foundations of Computer Science*, pp. 109–121.

Pratt, V. (1986), "Modelling Concurrency with Partial Orders", Report STAN-CS-86-1113, Department of Computer Science, Stanford University.

Reichgelt, H. (1987), "Semantics for Reified Temporal Logic", in *Advances in Artificial Intelligence* (Proc. AISB Conf., University of Edinburgh), J. Hallam and C. Mellish, eds., Wiley, pp. 49–61.

Reisig, W. (1985), *Petri Nets: An Introduction*, EATCS Monographs on Theoretical Computer Science , No. 4, Springer-Verlag, Berlin.

Rescher, N. and Urquhart, A. (1971), *Temporal Logic*, Springer-Verlag.

Rosenschein, S. J. and Kaelbling, L. P. (1986), "The synthesis of digital machines with provable epistemic properties", in *Proc. of the Conf. on Theoretical Aspects of Reasoning about Knowledge*, Joseph Y. Halpern, ed., Kaufmann, (An updated version appears as Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, California).

Rydeheard, D. E. and Burstall, R. M. (1988), in *Computational Category Theory*, International Series in Computer Science, Prentice Hall.

Rydeheard, D. E. and Stell, J. G. (1987), "Foundations of Equational Deduction: A Categorical Treatment of Equational Proofs and Unification Algorithms", in *Category Theory and Computer Science* (Proc. of the 2nd Conf., Edinburgh, September 1987), D. H. Pitt, A. Poigné and D. E. Rydeheard, eds., Lecture Notes in Computer Science , No. 283, Springer-Verlag, pp. 114–139.

Sacerdoti, E. D. (1974), "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence* 5, pp. 115–135.

Sacerdoti, E. D. (1977), *A Structure for Plans and Behavior*, Elsevier North-Holland Inc., New York.

Sannella, D. T. (1982), "Semantics, implementation and pragmatics of Clear: a program specification language", Ph.D. Thesis (Report CST-17-82), Department of Computer Science, University of Edinburgh.

Schmidt, D. (1984), "A Programming Notation for Tactical Reasoning", in *Proc. 7th Conf. on Automated Deduction*, R. E. Shostak, ed., Lecture Notes in Computer Science , No. 170, Springer-Verlag.

Schmidt-Schauß, M. (1985), "Unification in a Many-Sorted Calculus with Declarations", in *Proc. 9th German Workshop on Artificial Intelligence*, Informatik-Fachberichte , No. 118, Springer-Verlag, pp. 118–132.

Schmidt-Schauß, M. (1989a), in *Computational Aspects of an Order-Sorted Logic with Term Declarations*, Lecture Notes in Computer Science , No. 395, Springer-Verlag.

Schmidt-Schauß, M. (1989b), "Unification in a Combination of Arbitrary Disjoint Equational Theories", *Journal of Symbolic Computation* 8, pp. 51–99.

Schoppers, M. J. (1987), "Universal Plans for Reactive Robots in Unpredictable Environments", in *Proc. IJCAI 1987*, Vol. 2, pp. 1039–1046.

Shoham, Y. (1986), "Reified Temporal Logics: Semantical and Ontological Considerations", in *Proc. 7th ECAI*, Brighton, U.K..

Siekmann, J. H. (1989), "Unification Theory", *Journal of Symbolic Computation* 7, pp. 207–274, (Preliminary versions appear in *Proc. ECAI*, Vol. 2, 1986 and *Proc. 7th Conf. on Automated Deduction*, 1984).

Smithers, T. and Malcolm, C. (1988), "Programming Robotic Assembly in terms of Task Achieving Behavioural Modules", *Journal of Structural Learning* 10, pp. 137–156, (also, Research Paper 417, Department of Artificial Intelligence, University of Edinburgh).

Smolka, G. (1986), "Order-Sorted Horn Logic: Semantics and Deduction", SEKI report SR-86-17, Fachbereich Informatik, Universität Kaiserslautern.

Smolka, G. (1987), "TEL (Version 0.9): Report and User Manual", SEKI report SR-87-11, Fachbereich Informatik, Universität Kaiserslautern.

Smolka, G., Nutt, W., Goguen, J. A. and Meseguer, J. (1989), "Order-Sorted Equational Computation", in *Resolution of Equations in Algebraic Structures*, Hassan Ait-Kaci and Maurice Nivat, eds., Academic Press, (also, SEKI report SR-87-14, Fachbereich Informatik, Universität Kaiserslautern, 1987).

Sokolowski, S. (1983), "A Note on Tactics in LCF", Report CSR-140-83, Department of Computer Science, University of Edinburgh.

Steels, L. (1988), "Artificial Intelligence and Complex Dynamics", AI Memo 88-2, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, (presented at IFIP Workshop on *Concepts and Tools for Knowledge-Based Systems*, Mount Fuji, Japan, Nov. 1987).

Stoy, J. E. (1977), *Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass..

Stuart, C. J. (1985), "Synchronization of Multiagent Plans using a Temporal Logic Theorem Prover", Technical Note 350, A.I. Center, SRI International.

Stuart, C. J. (1986a), "Regular Expressions as Temporal Logic", Technical Report 64, Monash University, Clayton, Victoria, Australia.

Stuart, C. J. (1986b), *Branching Regular Expressions and Multi-Agent Plans*, Monash University, Clayton, Victoria, Australia.

Tate, A. (1976), "Project Planning using a Hierarchic Non-Linear Planner", Research Report 25, Department of Artificial Intelligence, University of Edinburgh.

Tennant, N. (1978), *Natural Logic*, Edinburgh University Press.

Tidén, E. (1986), "Unification in Combinations of Collapse-Free Theories with Disjoint Sets of Function Symbols", in *Proc. 8th International Conf. on Automated Deduction* , Oxford 1986, J. H. Siekmann, ed., Lecture Notes in Computer Science , No. 230, Springer-Verlag, pp. 431–449.

Valdes-Perez, R. E. (1986), "Spatio-Temporal Reasoning and Linear Inequalities", AI Lab. Memo 875, Massachusetts Institute of Technology.

Vere, S. (1981), "Planning in Time: Windows and Durations for Activities and Goals", Technical Report, NASA Jet Propulsion Laboratory.

Vijaykumar, R., Venkataraman, S., Dakin, G. and Lyons, D. M. (1987), "A Task Grammar Approach to the Structure and Analysis of Robot Programs", TR-87-67, Department of Computer and Information Science, University of Massachusetts at Amherst.

Vilain, M. and Kautz, H. (1986), "Constraint Propagation Algorithms for Temporal Reasoning", in *Proc. AAAI-86*.

Waldinger, R. (1977), "Achieving Several Goals Simultaneously", in *Machine Intelligence* , Vol. 8, pp. 94–136.

Wallen, L. A. (1985), "Generating Connection Calculi from Tableaux and Sequent Based Proof Systems", in *Proc. AISB Conf.* , Warwick, April 1985, (also, Research Paper 258, Department of Artificial Intelligence, University of Edinburgh).

Warren, D. H. D. (1974), "WARPLAN: A System for Generating Plans", Technical Report, Department of Artificial Intelligence, University of Edinburgh.

Warren, D. H. D. (1977), "Implementing Prolog – Compiling Predicate Logic Programs", Research Report 39, Department of Artificial Intelligence, University Of Edinburgh.

Wolper, P. L. (1982), "Synthesis of Communicating Processes from Temporal Logic Specifications", Ph.D. Thesis, Department of Computer Science, Stanford University.

Yarbus, A. L. (1967), *Eye Movements and Vision* (translated from Russian by Basil Haigh), Plenum Press, New York.

Yelick, K. A. (1985a), "Combining Unification Algorithms for Confined Equational Theories", in *Proc. 1st International Conf. on Rewriting Techniques and Applications* , Dijon, France, J-P. Jouannaud, ed., Springer-Verlag.

Yelick, K. A. (1985b), "A Generalized Approach to Equational Unification", Ph.D. Thesis (Report MIT/LCS/TR-344), Massachusetts Institute of Technology.

Yelick, K. A. (1987), "Unification in Combinations of Collapse-Free Regular Theories", *Journal of Symbolic Computation* 3, pp. 153–181.

Zachary, J. L. (1987), "A Framework for Incorporating Abstraction Mechanisms into the Logic Programming Paradigm", Ph.D. Thesis (Report MIT/LCS/TR-405), Massachusetts Institute of Technology.

Zachary, J. L. (1988), "A Pragmatic Approach to Equational Logic Programming", in *Logic Programming* , Proc. of the 5th International Conference and Symposium, Seattle, 1988, Robert A. Kowalski and Kenneth A. Bowen, eds., MIT Press, pp. 295–310.

Zhang, H. (1984), "REVEUR 4: Etude et mise en oeuvre de la réécriture conditionnelle", These de 3ème cycle, Univ. Nancy.

Glossary of Symbols

Index